SwiftCache User Manual

2013-03-15

v0.7.6-73-gf091255

Table of Contents

Table of Contents 2
1 Copyright and Confidentiality 9
1.1 Copyright Statement 9
1.2 Confidentiality Statement 9
2 Introduction 10
2.1 SwiftCache Overview 10
2.2 Terminology 10
3 Quick Start Guide 12
3.1 Overview 12
3.2 Network Setup 12
3.2.1 General Settings 13
3.2.2 Basic Network Settings 14
3.2.3 Advanced Network Settings 14
3.3 SwiftCache Setup 14
3.3.1 Proxy Settings 15
3.3.2 Disk Settings 16
3.3.3 License Settings 16
3.4 Logging Setup 17
3.4.1 Debug Level 19
3.4.2 Log Rotation 19
3.4.3 Log Upload 20
Specifying Log Filenames with Shell Wildcards 22
3.5 Starting and Stopping SwiftCache 22
3.6 Confirming Operation 22
4 SwiftCache Concepts 23
4.1 HTTP Caching 23
4.2 Hits and Misses 23
4.3 Hit Rates 23
4.4 Cache Index 23
4.5 License Options 23
5 Physical Installation 25
5.1 Technical Specifications 25
5.2 Racking Guidelines 25
6 Deployment Scenarios 26
6.1 Overview 26
6.2 Common Scenarios 26
6.2.1 Forward Proxy 26
6.2.2 Reverse Proxy 26
6.2.3 Important Considerations 26
6.3 Modes of Operation 26
6.3.1 Explicit Proxy 27
Avoiding Open Proxies 28
6.3.2 Semi-Transparent Proxy 29
6.3.3 Fully-Transparent Proxy 30
6.4 Deployment Topologies 31
6.4.1 Inline/Bridge Mode 31
6.4.2 Out of Path/Router Mode 32
6.4.3 Load Balancer in Bridge Mode 33

6.4.4 Load Balancer in Router Mode 34
6.4.5 Internet Service Provider (ISP) 35
Avoiding Asymmetric Routing 35
6.4.6 Reverse Proxy 36
6.5 Practical Deployment Considerations 36
6.5.1 Physical Connections 36
6.5.2 Pilot and Rollout 37
6.5.3 Sizing 37
6.5.4 Load Balancing 37
Compatible Layer 4-7 Switches and Load Balancers 37
Direct Connections from Load Balancers 37
7 SwiftCache User Interfaces 39
7.1 Overview 39
7.2 User Accounts 39
7.3 Configuration Daemon 39
7.4 Configuration Keys 39
7.4.1 Examples of Configuration Keys and Values 40
8 SwiftCache GUI 41
8.1 GUI Access 41
8.2 Logging In 41
8.2.1 SSL Warnings on GUI Login 42
8.3 Finding Help in the GUI 43
8.3.1 Contextual Help 43
8.4 Sections of the GUI 43
8.4.1 Home Tab (Dashboard) 43
Alert Summary 43
Performance Information 44
Status Table 44
8.4.2 Status Tab 45
Processes 45
Query Cache 46
TCP Stats 48
NIC Stats 49
Disk Stats 50
System Info 50
Cluster 51
Log 52
8.4.3 Config Tab 53
Changing Settings 54
8.4.4 Policies Tab 55
8.4.5 Filtering 57
Status 57
Test URL 57
Brightcloud 57
Global Lists 58
8.4.6 Reporting 58
Interacting with Graphs 58
Traffic 58
Network Throughput 59
Cache Throughput 60
Request Rates 61
Connection Stats 62
Top 100 62
Top Sites (Traffic) 63
Top Sites (Requests) 64
Top Clients (Traffic) 65
Top Clients (Requests) 66
System 67
Disk IO 68
Disk Usage 70
CPU Usage 71
Memory Usage 72
Performance 72
Bandwidth Savings 73
Hit Rates 74
Cache Status 75
Object Distribution 77
Service Time 78
Time To First Byte 78

SwiftSense 80
8.4.7 Alerts 80
Viewing and Acknowledging Alerts 80
Managing Notifications 80
8.5 Cluster Settings 81
8.5.1 Shared and Local Settings 81
8.5.2 Example 81
9 SwiftCache CLI 83
9.1 Overview 83
9.2 Applying Changes 83
9.3 CLI Usage 83
9.3.1 Running Commands 83
9.3.2 Tab Completion 83
9.3.3 CLI Help 84
9.3.4 Checking Configuration 85
Default Values 85
9.3.5 get and set Commands 85
9.3.6 add and remove Commands 85
9.3.7 Configuration Sections and Scope 86
9.3.8 create and no Commands 86
filter Sections 86
interface Sections 86
policy Sections 87
upstream_policy Sections 87
shared Section 87
local Section 87
9.4 Other CLI Commands 87
9.4.1 brightcloud 87
9.4.2 cache 88
9.4.3 cluster 88
9.4.4 config 88
9.4.5 edit 89
9.4.6 exit 89
9.4.7 idata 89
9.4.8 policy 89
9.4.9 process 90
9.4.10 raid show 91
9.4.11 reboot and shutdown 91
9.4.12 shell 91
9.4.13 ssl 91
9.4.14 stats 92
9.4.15 supercluster 92
9.4.16 test 93
9.4.17 top100 93
9.4.18 upgrade 93
9.4.19 upstream_policy 93
10 Operations Guide 95
10.1 Securing SwiftCache 95
10.1.1 User Administration 95
10.1.2 Best Practice Recommendations 96
10.2 Disk Replacement 96
10.3 Configuration Backup 96
10.3.1 GUI 96
10.3.2 CLI 96
11 Monitoring 97
11.1 Reporting 97
11.1.1 GUI 97
11.1.2 CLI 97

11.1.3 SNMP 97
11.2 Alerts 97
11.2.1 Alert Framework 97
11.2.2 Alert Workflow 98
11.2.3 Alert Suppression 99
11.2.4 Alert Priorities 99
11.2.5 Alert Details 100
11.2.6 Alert Notifications 102
12 Performance Tuning 104
12.1 Factors Affecting Performance 104
12.1.1 Request Rate 104
12.1.2 Simultaneous Connections 104
12.1.3 Cache Efficiency 104
12.1.4 Client Speed 104
12.1.5 Filtering 104
12.1.6 Policies 104
12.1.7 Disk IO 105
12.2 Improving Performance 105
12.2.1 Maximising Bandwidth Savings 105
SwiftSense 106
13 Clustering 107
13.1 Overview 107
13.2 Cluster Configuration 107
13.2.1 Adding a New Appliance to a Cluster 107
13.2.2 Removing a Node from a Cluster 108
13.2.3 Viewing Cluster Status 109
13.3 Superclusters 110
13.3.1 Supercluster Setup 110
GUI 110
CLI 110
13.4 Automated Cluster Upgrades 110
13.4.1 Automated Upgrade Workflow 110
13.4.2 Failures During Upgrade 111
13.4.3 Deleting Old Software Versions 111
13.4.4 Important Warnings 111
13.5 Inter-Cache Communication (ICC) 111
13.5.1 ICC Setup 112
13.5.2 Request Workflow Without ICC 112
13.5.3 Request Workflow With ICC 113
13.5.4 Advanced Information 113
Items Not Handled by ICC 113
Effect of Node Leaving an ICC Cluster 113
Caches Must Retrieve an Item Once 113
14 Policies 114
14.1 Overview 114
14.2 Introducing Policies 114
14.3 SwiftSense Policies 114
14.3.1 Management 114
14.3.2 Recommendations 116
14.4 Custom Policies 116
14.4.1 Filter Policies 116
14.4.2 Ordering of Policies 117
14.4.3 Match Rules 117
Named Captures 118
14.4.4 Common Settings 118
cache_partial_download 118
complete_download 118
cache_async_fetch 119

cache_async_refresh 119
ignore_range 119
14.4.5 Video Seek 119
14.4.6 Safe Search 120
Google 120
Bing 120
15 Filtering 121
15.1 Overview 121
15.2 White Lists 121
15.3 Black Lists 121
15.4 Global Black and White Lists 121
15.4.1 Global White List 122
15.4.2 Global Black List 122
15.5 Filter Policies 122
15.5.1 Terminology 122
15.5.2 Configuration 123
15.5.3 Filter Status and Testing 124
15.6 Brightcloud (Dynamic URL Classification) 125
15.6.1 Categories 126
15.6.2 Status and Information 126
15.6.3 Usage 126
16 Advanced Features 127
16.1 Overview 127
16.2 DNS Resolver 127
16.3 Asymmetric Routing 127
16.3.1 Enabling Asymmetric Mode 127
16.4 Fast Disks 128
16.4.1 Improving Disk Seek Time 128
16.4.2 Enabling Fast Disks 128
16.5 Trust X-Forwarded-For HTTP Header 128
16.5.1 Enabling Trust X-Forwarded For 128
16.6 Return to Sender 128
16.6.1 Enabling Return to Sender 129
16.7 IPv6 Support 129
16.8 Overload Protection 129
16.8.1 Bypass Mode 129
16.8.2 Relay Mode 130
16.9 SSL Support 130
16.9.1 SSL Proxy Mode 130
Distribution of SSL Certificates 130
Re-establishing a Chain of Trust 131
Valid Certificate Workflow 131
Invalid Certificate Workflow 131
Enabling SSL Proxy Mode 131
SSL Proxy Mode and Overload Protection 132
16.9.2 SSL Relay Mode 132
16.10 Limiting Download Rates 132
16.10.1 Enabling Rate Limiting 132
17 SwiftSense 133
17.1 Overview 133
17.2 Security 133
17.3 Functions 133
17.4 SwiftSense User Interfaces 134
17.4.1 SwiftCache GUI 134
17.4.2 SwiftSense Web Interface 134
17.4.3 Dashboard 134

17.4.4 Caches 135
17.4.5 Organisations 135
17.4.6 Reporting 135
17.4.7 System 135
17.4.8 Admin 136
18 Troubleshooting 137
18.1 Diagnostics 137
18.2 Support 137
19 Appendix A: Log File Format 138
Log File Format 138
Log Status 140
Log Info 140
Filtering Info 145
20 Appendix C: Configuration Key Reference 147

1 Copyright and Confidentiality

1.1 Copyright Statement

© Copyright SwiftServe Pte Ltd, 2013. All rights reserved.

No part of this documentation may be reproduced in any form or by any means or be used to make any derivative work (including translation, transformation or adaptation) without explicit written consent of SwiftServe Pte Ltd.

Registered address: 8 Temasek Boulevard, Suntec Tower 3, #20-01, Singapore 038988

Company Registration No.: 201019734M

1.2 Confidentiality Statement

All information contained in this documentation is provided in commercial confidence for the sole purpose of adjudication by SwiftServe Pte Ltd and Customer/Partner. The pages of this document shall not be copied published or disclosed wholly or in part to any party without SwiftServe Pte Ltd prior permission in writing, and shall be held in safe custody.

2 Introduction

2.1 SwiftCache Overview

SwiftCache is a high-performance caching appliance that has been designed primarily for the carrier and ISP marketplace. It has been built using a modular and flexible platform, which means it can be deployed in varying scenarios such as ISP, enterprise, corporate or content delivery networks.

The SwiftCache architecture is optimised to cache all types of HTTP and web traffic including video and multimedia content. SwiftCache features a flexible configuration language allowing the definition of customised policies so that popular websites can take full advantage of the content caching.

SwiftCache has optional add-ons available for caching and handling RTMP traffic and caching SSL (HTTPS) traffic in enterprise deployments.

Installing SwiftCache within a network will typically reduce the amount of transit or external bandwidth required by between 25 percent and 80 percent, depending on the traffic profile and how the platform is deployed, configured, integrated and tuned.

A typical SwiftCache deployment within an ISP network would consist of between two and forty caches in one or more locations. Within each cache location the caches (or cluster *nodes*) would be joined together to form a SwiftCache *cluster* where the majority of configuration is identical and shared between them; all the SwiftCache appliances would be integrated into the network together to provide a complete caching service.

A deployment scenario within an enterprise network could start with just a single SwiftCache installed to reduce an organisation's external bandwidth usage and to control and monitor users' browsing habits.

2.2 Terminology

The following terms and conventions are used in this document:

Origin Server	An HTTP web server that provides the source of the web content to be cached (e.g. www.bbc.com).
Item or *Object*	A specific piece of content requested via HTTP from a web server (e.g. an image, web page or video).
Appliance	A hardware device running integrated software for a specific purpose (e.g. caching web content).
SwiftCache or *Cache*	A SwiftCache appliance.
Proxy	Caching software running on SwiftCache.
User or *End-User*	The person viewing the web content served via SwiftCache from the origin server (e.g. a customer of an ISP).
Client	The software or application used by the end-user to request and view web content (e.g. a web browser).
Operator	The administrator of the SwiftCache appliance.
Cluster	A group of SwiftCache appliances configured to communicate and operate together as a single unit.
Node	A single SwiftCache appliance within a cluster of SwiftCaches.

3 Quick Start Guide

3.1 Overview

This chapter describes the typical steps to take for the initial setup of a SwiftCache appliance. It is intended as a quick reference guide for operators already familiar with SwiftCache.

If you are not familiar with SwiftCache operation, please read the chapter on **SwiftCache Concepts** and following chapters first.

3.2 Network Setup

Depending on your vendor, your SwiftCache may be delivered with the network already configured, otherwise it will be in the default DHCP mode.

To begin configuring the SwiftCache you will need to know its initial assigned IP address. If preconfigured by your vendor, please consult their documentation.

If SwiftCache is running in DHCP mode, please establish its assigned IP address by querying the DHCP server on your network or by consulting your network administrator.

Open a web browser on a device connected to the same network as the SwiftCache appliance and go to:

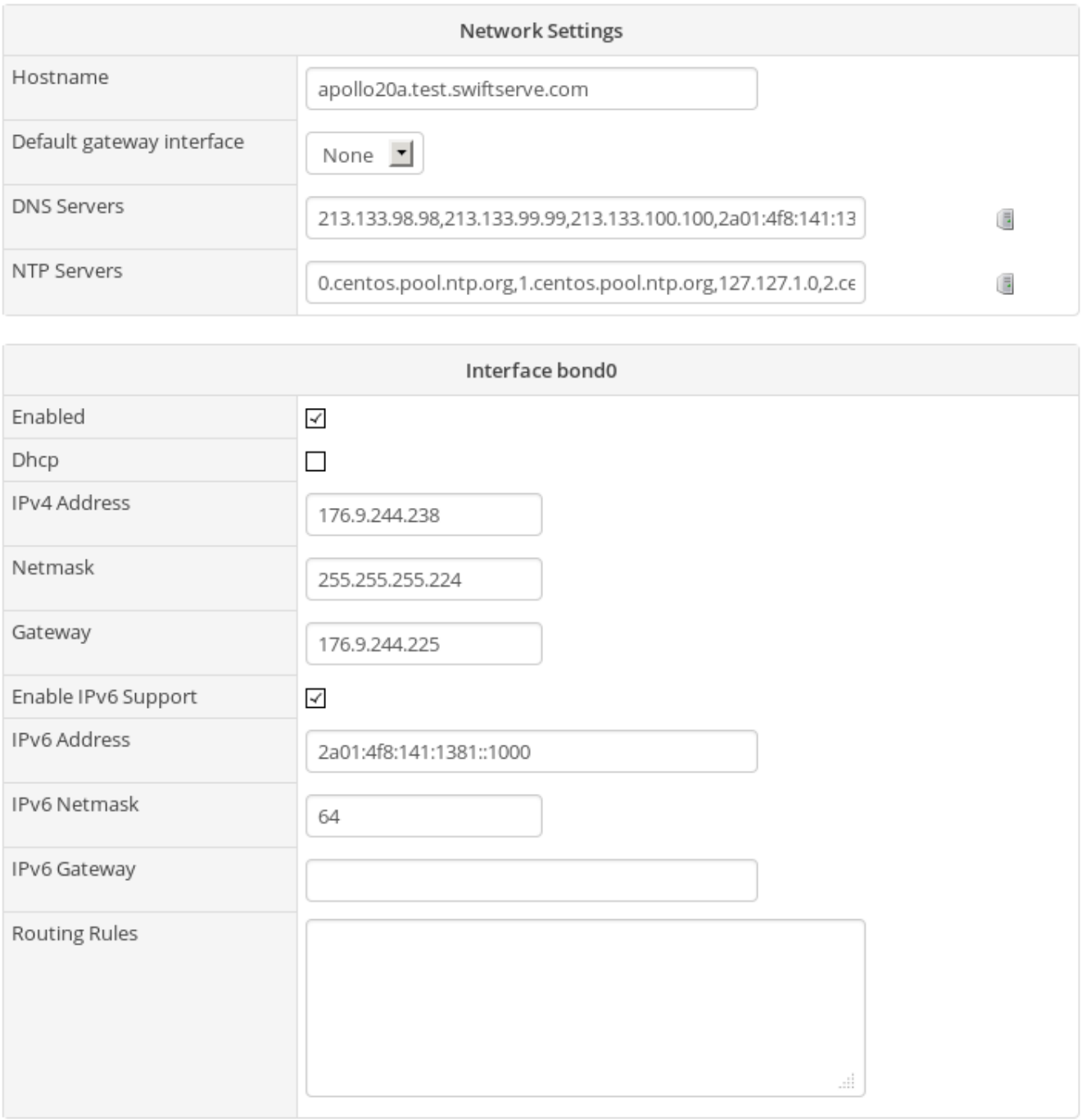
```
http://<cacheip>:8500
```

replacing `<cacheip>` with the IP address of your appliance.

Enter the username `admin` and the password supplied by your vendor to log in. For more details on connecting to the GUI, please refer to the **SwiftServe GUI** chapter of this document.

Click on the *Config* tab and then the *Network* subsection to view and edit the current settings. Note that network settings are applied immediately after clicking *Update*. It may take a few moments for new network settings to take effect while the networking subsystem is restarted on the SwiftCache appliance.

Care should be taken when adjusting network settings to avoid losing contact with the device.



Network settings are grouped into three areas:

General	These are the basic settings that are not specific to any individual network interface.
Bridged Interface	A bridged interface (br0) is defined by default on every new install.
Other Interfaces	These are all of the physical network interfaces on the machine.

3.2.1 General Settings

Hostname	The system hostname.
DNS Servers	List of DNS nameservers to use.
NTP Servers	Network time servers to use.

3.2.2 Basic Network Settings

Enabled	Whether the specific interface should be enabled or disabled.
IPv4 Address	IPv4 address to apply to the network interface.
Netmask	IPv4 network mask to apply to this interface.
Gateway	IPv4 gateway address for this interface.
Enable IPv6 Support	Whether IPv6 should be enabled or disabled on this interface.
IPv6 IP Address	IPv6 address to apply to the network interface.
IPv6 Netmask	IPv6 network mask to apply to this interface.
IPv6 Gateway	IPv6 gateway address for this interface.

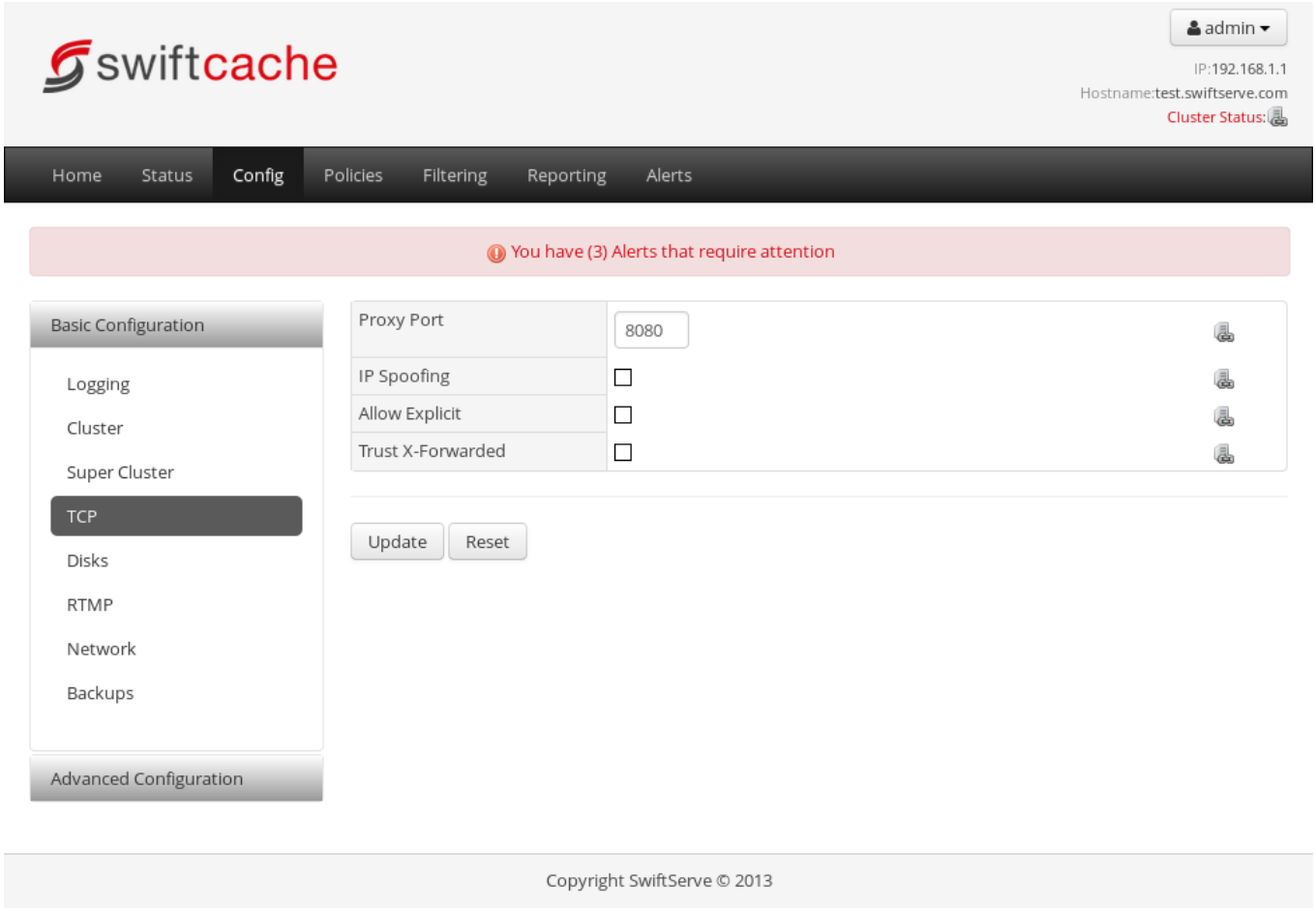
3.2.3 Advanced Network Settings

Auto Negotiation	Ethernet procedure to define appropriate interface transmission parameters. This should remain enabled unless interface errors are experienced.
Speed	The interface transmission speed. Required if Auto Negotiation is disabled on the interface.
Duplex	Duplex mode for the interface (either half or full). Required if Auto Negotiation is disabled on the interface.
Master Interface	This setting is used to make current interface part of a bridge or for interface bonding. If set, parameters like th IP addresses and netmasks are ignored on the interface. This setting is not applicable to non-Ethernet interfaces.
Routing Rules	Static routing rules for the interface.

3.3 SwiftCache Setup

Once the network has been configured, the proxy behaviour, disk settings and license key then need to be checked and configured before deployment. Depending on your vendor, it is possible that your SwiftCache may have been shipped with one or more of these already configured.

3.3.1 Proxy Settings



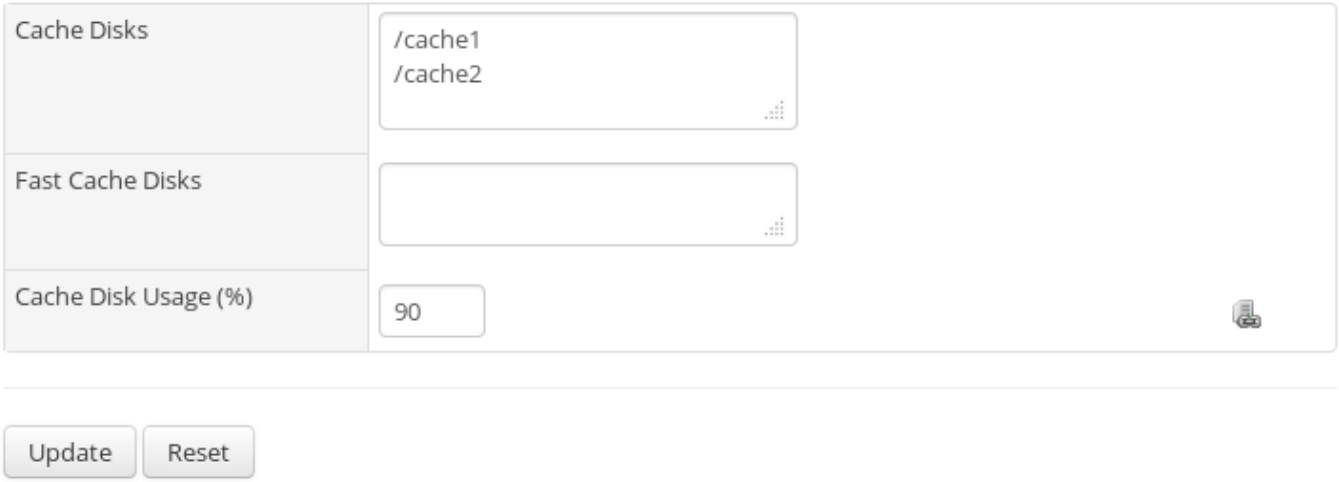
The proxy port and parameters are specified on the *TCP* subsection of the *Config* tab in the GUI. These settings controls the main operation of the caching software and typically only need to be modified once on installation. The settings chosen will depend on the mode of operation chosen.

For more information on possible modes of operation and the settings below, please refer to the **Deployment Scenarios** chapter.

Proxy Port	TCP port that the proxy process should bind to and listen for incoming HTTP requests (Default: 8080).
IP Spoofing	Controls whether the SwiftCache should spoof the client IP address when deployed in a fully-transparent mode of operation.
Allow Explicit	Controls whether the SwiftCache should allow explicit (direct) connections.
Trust X-Forwarded	Controls whether the SwiftCache should trust the `X-Forwarded-For` HTTP request header and use it as the client IP address for the purposes of evaluating policies and IP spoofing.

To avoid SwiftCache being used as an open proxy it is strongly recommended that the *Allow Explicit* setting is disabled in any production environment, or that explicit mode is only used when access is restricted only to authorised client IP addresses.

3.3.2 Disk Settings



The cache disks used by the device are specified on the *Disks* subsection of the *Config* tab. Here the operator can control the disks used and configure the proportion of the available disk space to use for cached content.

Cache Disks	A list of mount points for the cache disks within the Swiftcache. Typically these are named `/cache1` to `/cache10`.
Fast Cache Disks	A list of fast cache disk mount points for some content. Leave this blank if all the cache disks have the same speed.
Cache Disk Usage	Disk utilisation threshold. When usage on any individual disk reaches this threshold the cache cleaning process is triggered to bring usage below the threshold by removing old objects from the cache.

3.3.3 License Settings

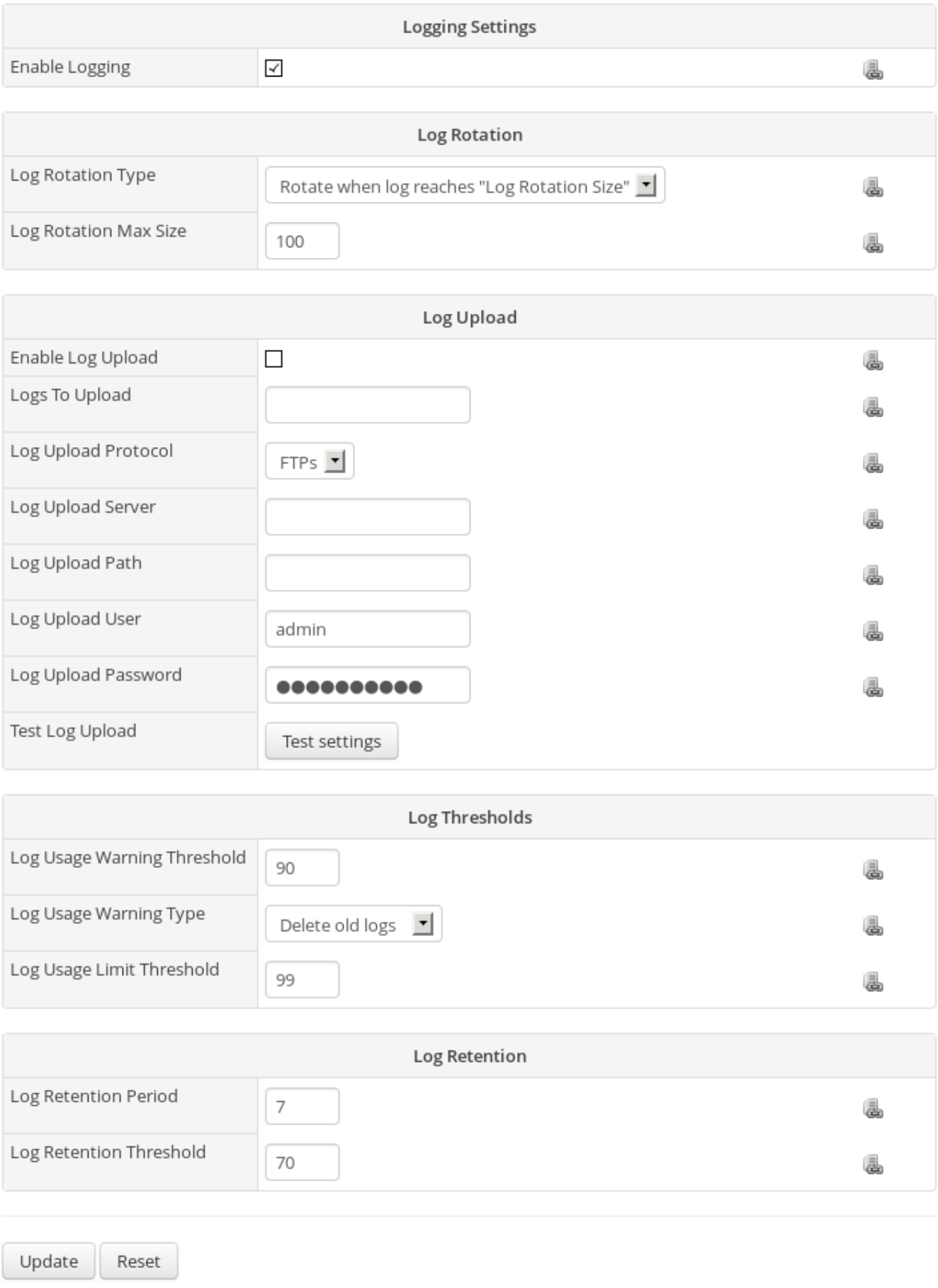
A SwiftCache license key is time-limited and may only be used with its intended appliance. A license key cannot be transferred from one SwiftCache to another. Without a valid license key the SwiftCache proxy software will not start.

The license key will usually have been set by the vendor before dispatch. You can confirm this by looking at the *License* subsection of the *Config* tab, under *Advanced Configuration*.

If a SwiftCache appliance or its hardware components are replaced, a new license key will be required in place of the old key. In this situation, it may be necessary to send the license information of your old key to your vendor in order to regenerate the key.

To apply a new license key, click on the *Edit* button on the right-hand side, then enter the new key before clicking on *Save*. This will write the license key to the `/usr/local/cache/license.key` file on the SwiftCache.

3.4 Logging Setup



SwiftCache provides powerful capabilities to manage the log files produced by the appliance. Two types of log files are produced:

`access.log`	This file records all requests processed by the SwiftCache and contains additional information to record cache and filtering info. Please refer to **Appendix A** for an overview of the SwiftCache access log format.
`error.log`	This file contains diagnostic information from the SwiftCache. Please refer to the appendix for a discussion of some of the key entries in this log file.

The log settings used by the device are specified on the *Logging* subsection of the *Config* tab.

Enable Logging	Whether to enable or disable all logging completely.
Debug Level	Diagnostic level from 1 to 5, default: 2.

3.4.1 Debug Level

SwiftCache provides a configurable diagnostic level to control the level of detail that is written to the error log:

- Level 1: Errors [E]
- Level 2: Warnings [W]
- Level 3: Information [I]
- Level 4: Debug [D]
- Level 5: Trace [T]

These levels are cumulative; for example, Level 2 will contain both errors and warnings.

Note that it is not recommended that the diagnostic level is increased beyond Level 3 for more than a short period. This is due to the very large volume of log lines that will be generated, especially on a heavily-loaded SwiftCache.

3.4.2 Log Rotation

SwiftCache archives log files according to their size, age, or at a set time according to a schedule. Archived log files are rotated and compressed to reduce the amount of disk space utilised.

Log Rotation Type	Trigger to rotate and compress log files. Three triggers are supported: when a log reaches a certain size; periodic rotation after X hours; or scheduled rotation at the same time every day.
Log Rotation Max Size	Maximum size in MB of an individual log file after which it should be rotated. This setting is only applied if *Rotate when log reaches* '*Log Rotation Size*' is selected.
Log Rotation Period	Maximum age of a log file in hours. This setting is only applied if *Periodic* rotation is selected.
Log Rotation Schedule	Time at which log files should be rotated. This setting is only applied if *Scheduled* rotation is selected.

3.4.3 Log Upload

Optionally log files may be uploaded to a remote log server to free up space on the SwiftCache, otherwise SwiftCache will delete the oldest archived log files if the log partition fills up.

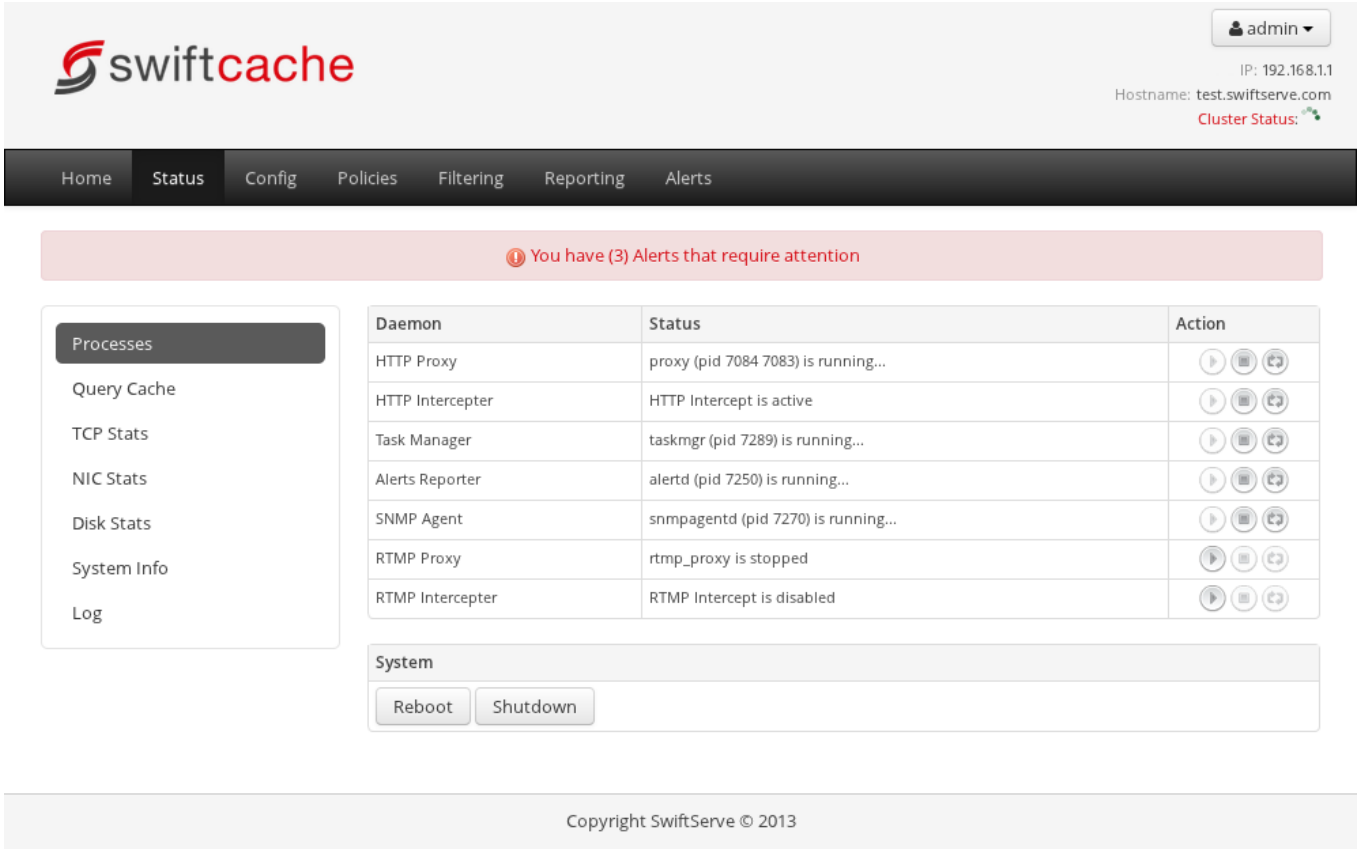
Enable Log Upload	Whether to upload log files to a remote log server immediately after they have been rotated and compressed.
Logs To Upload	Comma-separated list of log filenames to upload. Log filenames use the following format: `<logname>-<hostname>-<starttime>-<endtime>.log.gz`. An empty value means upload everything. Filenames may also be specified using shell wildcards (see below).
Log Upload Protocol	File transfer protocol to use when uploading log files: either FTP, SFTP or FTPs.
Log Upload Server	Full hostname or IP address of log server.
Log Upload Path	Path to a folder on the log server to which the logs should be uploaded.
Log Upload User	Username that SwiftCache should use to authenticate with log upload server.
Log Upload Password	Password that SwiftCache should use to authenticate with log upload server.
Log Usage Warning Threshold	The percentage of log partition usage that will trigger an action.
Log Usage Warning Type	The action to perform if the threshold is reached.
Log Usage Limit Threshold	The percentage of log partition usage which will cause logging to be disabled.
Log Retention Period	The maximum age of retained log files (days). Zero means never delete.
Log Retention Threshold	Percentage of log partition size to retain log files.

Specifying Log Filenames with Shell Wildcards

Shell wildcards can be used to specify multiple log filenames. Shell wildcards are `*`, `?`, `[seq]`, `[!seq]`.

For example, `access*.gz` would specify that all log filenames starting with `access` and ending with `.gz` should be uploaded.

3.5 Starting and Stopping SwiftCache



The core SwiftCache daemons will start automatically when the system is powered on. It is possible to view and control the current state of the individual processes through the *Processes* subsection of the *Status* tab in the GUI. Processes may be started, stopped and restarted.

To reboot or shut down and power off the SwiftCache appliance, click on the *Reboot* or *Shutdown* button. Care should be taken not to shut down the system inadvertently or when there is nobody physically present to power up the appliance again.

3.6 Confirming Operation

It is possible to quickly validate that a SwiftCache is active and serving traffic by looking at the *Home* tab in the GUI.

To begin with it would expected to see some network throughput (blue line) on the graph. Once the appliance has been operating for some time there would be some bandwidth savings (red line) also.

After several days' normal usage it would expected that the red line would be above the blue line.

4 SwiftCache Concepts

4.1 HTTP Caching

A HTTP *cache* (or *web proxy*) such as SwiftCache connects to web servers (*origin servers*, because they hold the original content) and retrieves items (or *objects*) on behalf of end-users' *client* systems, such as a web browser.

When possible, the cache will store a copy of the object locally so that, if the same or other clients request the object again, the cache will be able to return it to the clients without having to request the object again from the origin server.

Caching items will improve the user experience as it typically speeds up the delivery of the item. This benefit is particularly apparent when the cache is close to the end-user and the origin server is distant. This is because the client request only has to travel to the cache and back, not all the way to and from the origin server.

For the content provider running the origin server, caches will also reduce the external bandwidth used because fewer requests have to come to the origin server.

Over time a cache will build up a local copy of the most frequently-accessed content allowing the majority of requests to be accelerated. The cache checks objects periodically for freshness and expires (removes) old ones from the cache. Should the item then be requested again, the cache will retrieve a fresh copy from the origin server again.

Sometimes the client or origin server may specify an object as uncacheable, however policies defined on the cache can override this behaviour if desired.

4.2 Hits and Misses

A *cache hit* happens when a client requests an object already stored by the cache; it can be served from the cache without requiring it to be retrieved from the origin server.

A *cache miss* occurs when a client requests an object which is not already stored by the cache; it is necessary to retrieve it from the origin server. It may optionally be saved for later use depending on the usage patterns and configured parameters, as well as the server and client HTTP headers.

4.3 Hit Rates

Hit rates are a measure of how successful a cache is being. The *request hit rate* is the proportion of connections that result in a cache hit. The *byte hit rate* is the proportion of the overall data traffic that is delivered by the cache. A higher hit rate is better.

4.4 Cache Index

SwiftCache keeps a list (or *cache index*) of all the objects that it has stored. Client requests for content are compared against this list to determine if the item is available to be served from the local cache.

To ensure a high cache hit rate, it is important to avoid the same object being indexed multiple times. This may happen when different URLs are used to request the same object, such as when an origin server dynamically adds unique session identifiers into the URL.

4.5 License Options

There are a number of optional, add-on modules available to the basic SwiftCache license:

Brightcloud	Dynamically identifies the web category of content being requested. See the **Brightcloud (Dynamic URL Classification)** section in the **Filtering** section of this manual for more information.
Fast Disks	Enables the use of Solid State Disks (SSDs) to boost cache performance. See the **Fast Disks (SSD)** section in the **Advanced Features** section for more information.
RTMP	Enables the caching and management of RTMP traffic in addition to HTTP traffic.
SSL Proxy	Enables the caching and management of SSL-encrypted traffic. See the **SSL Caching** section in the **Advanced Features** section for more information.
SwiftCDN	Allows the SwiftCache appliance to be connected to SwiftServe's global Content Delivery Network (CDN) service, enabling the appliance owner to generate revenue from delivering content on behalf of SwiftServe.

5 Physical Installation

5.1 Technical Specifications

Please see the separate documentation provided with your SwiftCache appliance.

5.2 Racking Guidelines

Please see the separate documentation provided with your SwiftCache appliance.

6 Deployment Scenarios

6.1 Overview

To gain the best possible performance from SwiftCache, it is important to ensure that it is deployed into the best location on a network. However, due to the variety of possible network configurations, the optimal location for SwiftCache will vary depending on the specific environment into which it is being deployed.

6.2 Common Scenarios

6.2.1 Forward Proxy

A common deployment scenario is where SwiftCache is deployed to reduce the amount of traffic travelling from within an provider's network to the Internet.

This in turn would reduce bandwidth costs and potentially improve performance on congested network links.

In this scenario SwiftCache can be positioned at the network edge, border or core for use by clients within the network, for example with a separate SwiftCache cluster at each network gateway. This is sometimes referred to as a *forward proxy*.

6.2.2 Reverse Proxy

It may also be important to reduce the amount of traffic entering an organisation's network. This may be when clients outside of the organisation's network are requesting the same content repeatedly from origin servers within the network.

In this scenario, it is recommended to deploy SwiftCache in multiple locations. These are usually at the edge of the core network or even in the metro network. SwiftCache will respond to client requests on behalf of the origin servers. This is sometimes referred to as a *reverse proxy*.

6.2.3 Important Considerations

Irrespective of where SwiftCache is positioned within a network, the most important considerations for any SwiftCache deployment are to ensure that it is resilient to failure and that network traffic is routed symmetrically through it.

6.3 Modes of Operation

When deploying SwiftCache into a network, it is important to first consider:

- how client requests, such as those from a web browser, will be routed to SwiftCache; and
- whether the end-user will be aware that requests are being proxied through SwiftCache.

It is not recommended simply to place SwiftCache directly in the main path of all client requests. This would cause all network traffic, not just web traffic, to pass through SwiftCache. This approach would be high-risk, inefficient and would scale poorly.

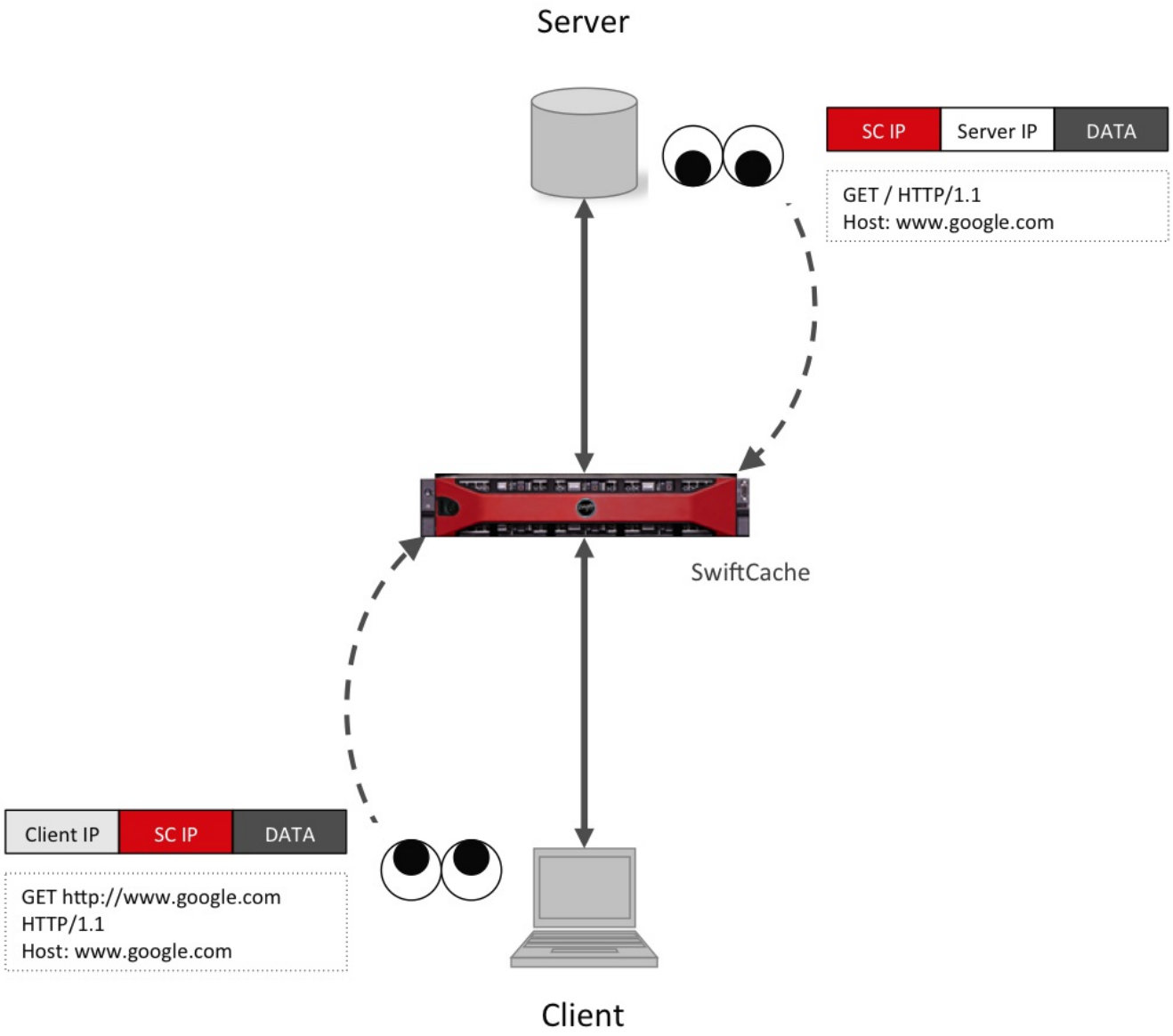
Instead, SwiftCache supports three modes of operation:

- *explicit proxy*;
- *semi-transparent proxy*; and

- *fully-transparent proxy*.

These are discussed in more detail below.

6.3.1 Explicit Proxy



- Client is aware of the SwiftCache
- Client sends request to the SwiftCache instead of the Server
- Explicit GET request contains entire URL

Using SwiftCache as an explicit proxy relies on the end-user configuring their client software to route outbound web requests through SwiftCache.

For example, most web browsers support the ability to define the location of a web proxy.

This type of deployment used to be common in enterprise and corporate networks where the organisation

required a forward proxy to control its employees' access to the Internet. However this approach is becoming less popular with businesses as it places a large support overhead on the internal IT function to ensure that all clients are correctly configured to use the proxy.

Similarly, this approach is impractical within an ISP-scale deployment as the ISP does not control the end-user's devices and clients and so cannot ensure they are correctly configured to route requests through the web proxy. Instead, transparent proxy deployments are becoming more popular in both corporate and ISP environments because they remove the need for changes to the client configuration in exchange for some additional network complexity.

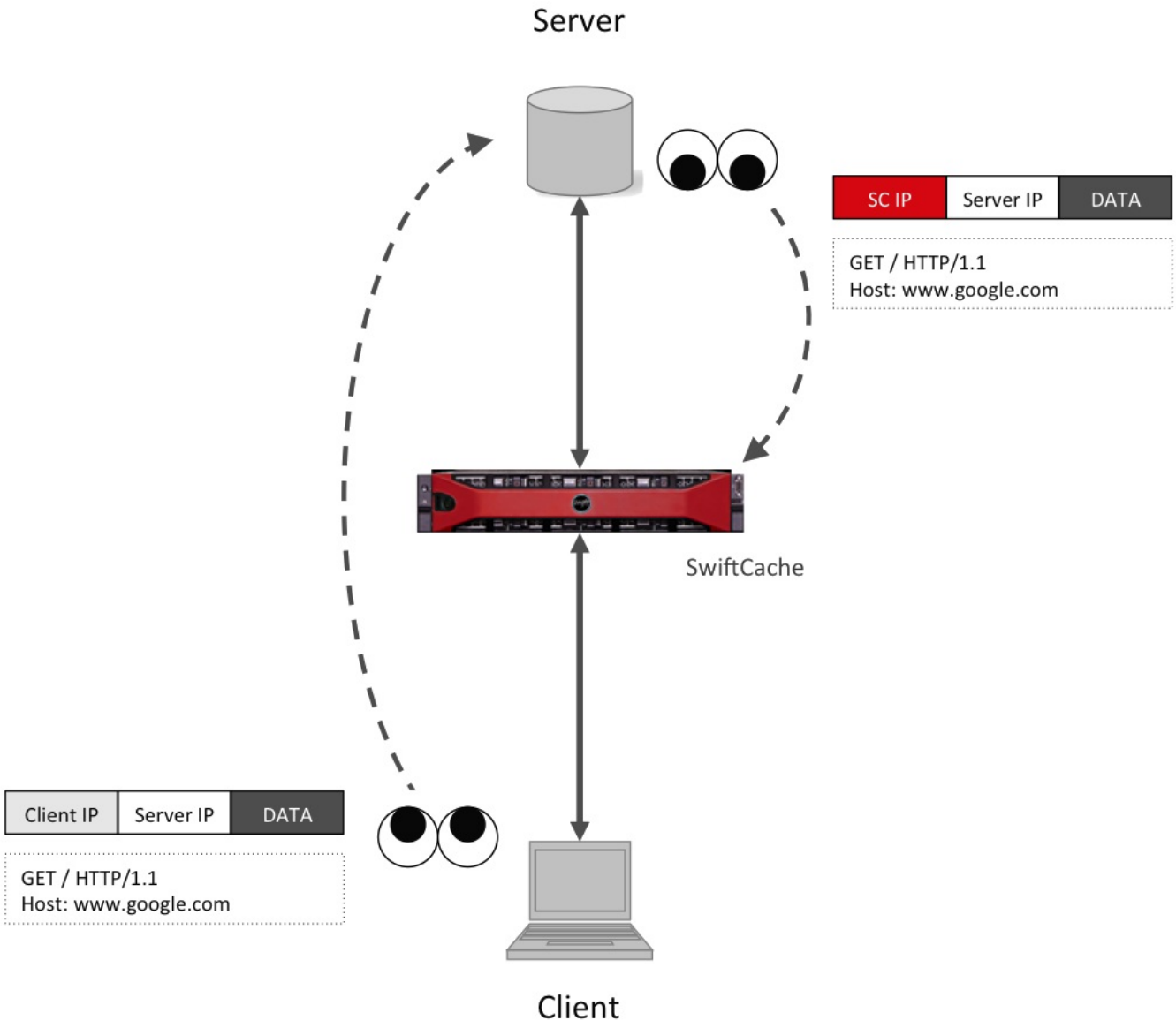
Avoiding Open Proxies

Explicit mode can be useful for testing purposes, as it does not require any specific network configuration. However this mode of operation can be extremely dangerous if enabled in a production environment without any access restrictions.

To do so would create an *open proxy*, a web proxy that allows any client to request content from any origin server. Because this effectively anonymises the client requests (all requests to the origin server would appear to come from the open proxy, not from the client), open proxies can be used to avoid filtering and monitoring, and are often exploited to conduct malicious or illegal activities.

To avoid SwiftCache being used as an open proxy it is strongly recommended that explicit mode is only used when access is restricted only to authorised client IP addresses.

6.3.2 Semi-Transparent Proxy



- Client is unaware of the existence of the SwiftCache
- Server sees the IP of the SwiftCache
- Easy to deploy. The response finds its way back to the SwiftCache

Using SwiftCache in a semi-transparent mode of operation relies on Policy-Based Routing (PBR) to be configured in the network to intercept outbound client requests and direct them to SwiftCache in a transparent manner.

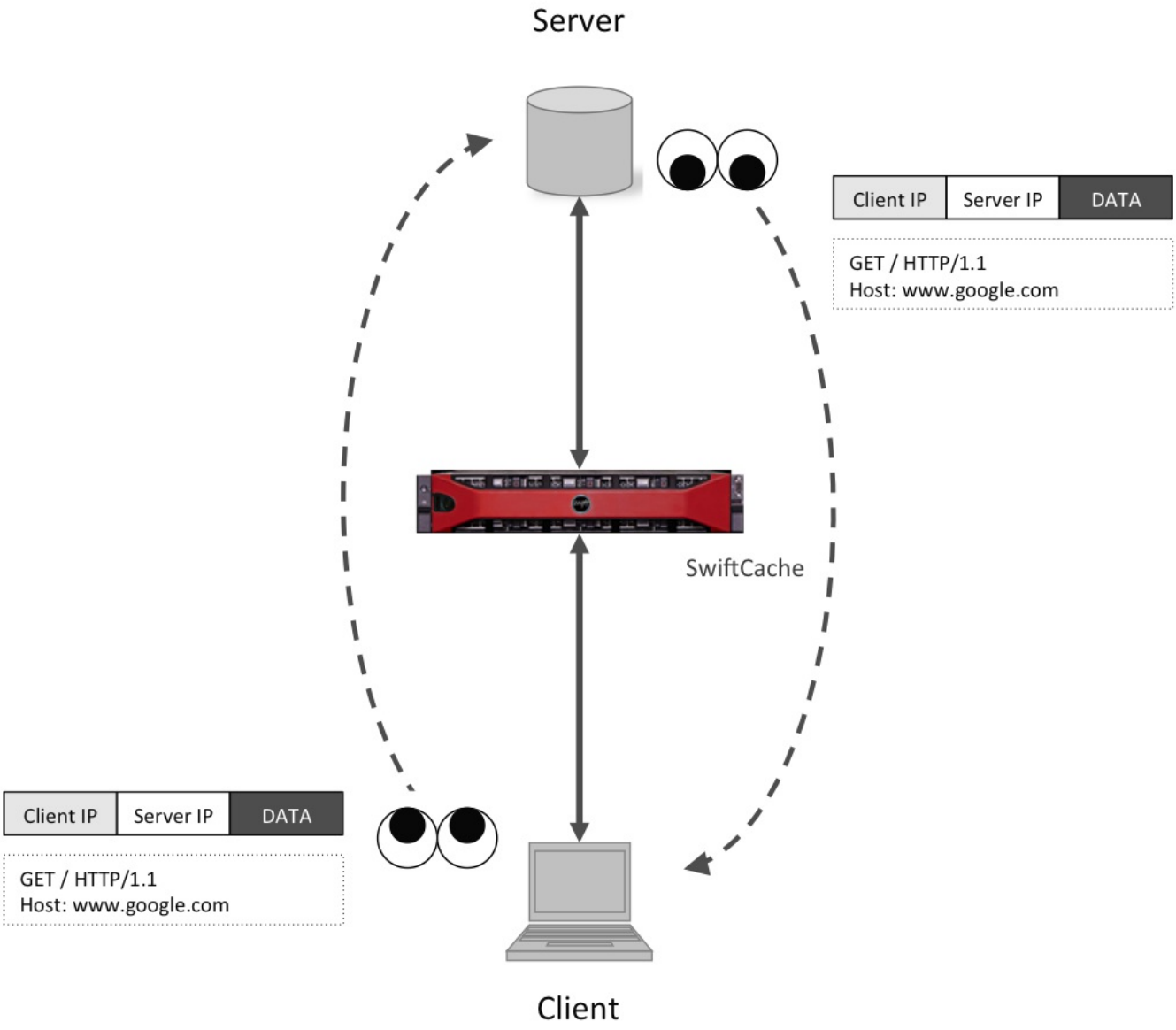
The end-user is unaware that their requests are being routed to SwiftCache and so this approach removes the need for any client configuration changes. However, this mode is semi-transparent because SwiftCache remains visible to the origin server, which will see requests coming from the IP address of SwiftCache rather than the client.

In some cases this mode of operation can break application logic on the origin server. For example, some file-

sharing websites may deliberately delay multiple, simultaneous downloads from the same end-user in a “waiting room” based on the client's IP address.

In a semi-transparent deployment, requests from multiple clients via the SwiftCache will all appear to be coming from the same IP address. The origin server will interpret this as a single end-user requesting many parallel downloads, when in reality it is multiple end-users, and incorrectly slow down the connections.

6.3.3 Fully-Transparent Proxy



- Client is unaware of the existence of the SwiftCache
- SwiftCache spoofs the Client IP in the request to the Server
- Server sees the IP of the Client and responds to it
- Note: A Router needs to intercept both the Client's HTTP request and the Server's HTTP response and redirect to the SwiftCache

Using SwiftCache in a fully-transparent mode of operation resolves the problem noted above, in which the origin server sees requests coming from the IP address of SwiftCache rather than the client. In this mode, SwiftCache is now transparent to both the client and the origin server.

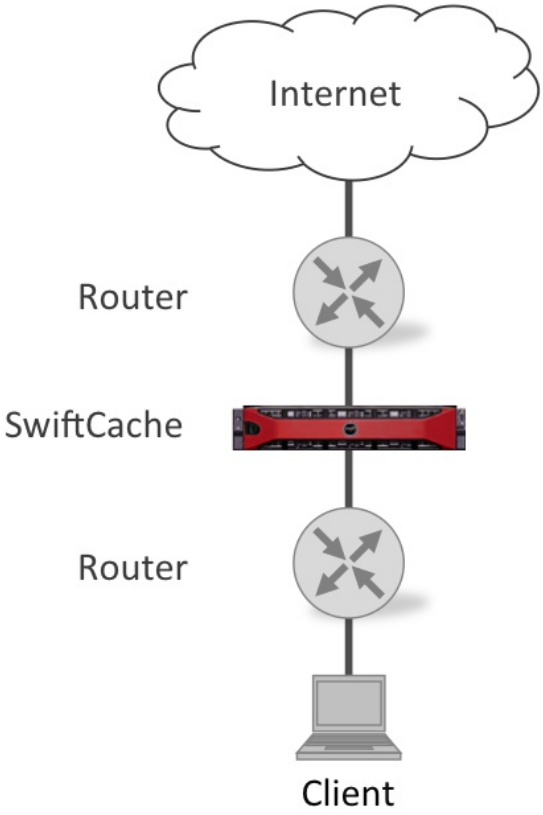
When fully-transparent mode is enabled, SwiftCache will rewrite the source IP address in network packets sent to the origin server so that it appears as if the request was sent directly from the client. This is sometimes referred to as *IP spoofing*. This is the recommended method of operating SwiftCache in a standard network environment.

6.4 Deployment Topologies

This section describes typical network topologies for SwiftCache deployments:

- Inline/Bridge Mode
- Out of Path / Router Mode
- Load Balancer in Bridge Mode
- Load Balancer(s) in Router Mode
- Internet Service Provider
- Reverse Proxy Mode

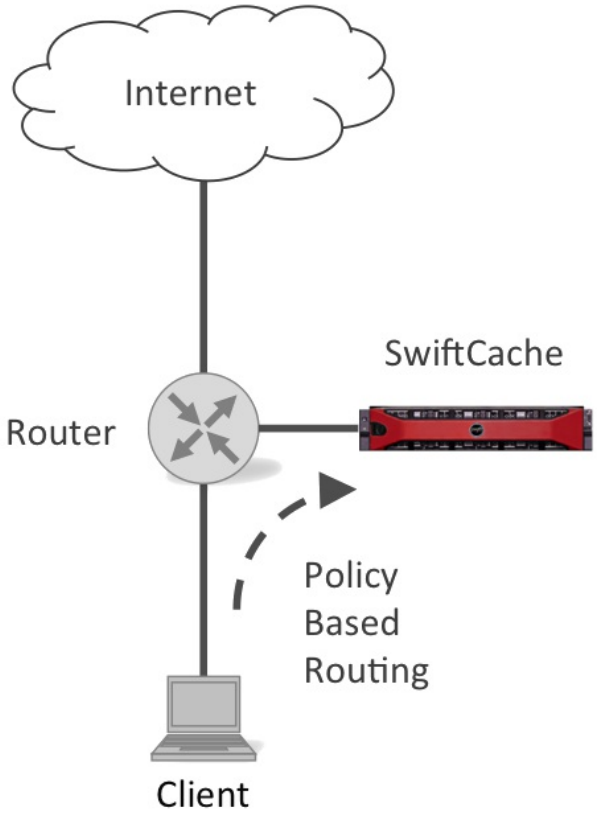
6.4.1 Inline/Bridge Mode



This mode places the SwiftCache inline using *bridged* networking. It has the advantage of quick deployment and simple routing, but is not scalable. All traffic passes through the cache, so a failure would lead to a complete loss of service. The SwiftCache can be configured with a fail-to-wire network interface card (NIC) to prevent any outage in such instances.

An example where inline/bridge mode may be suitable is a small office setup.

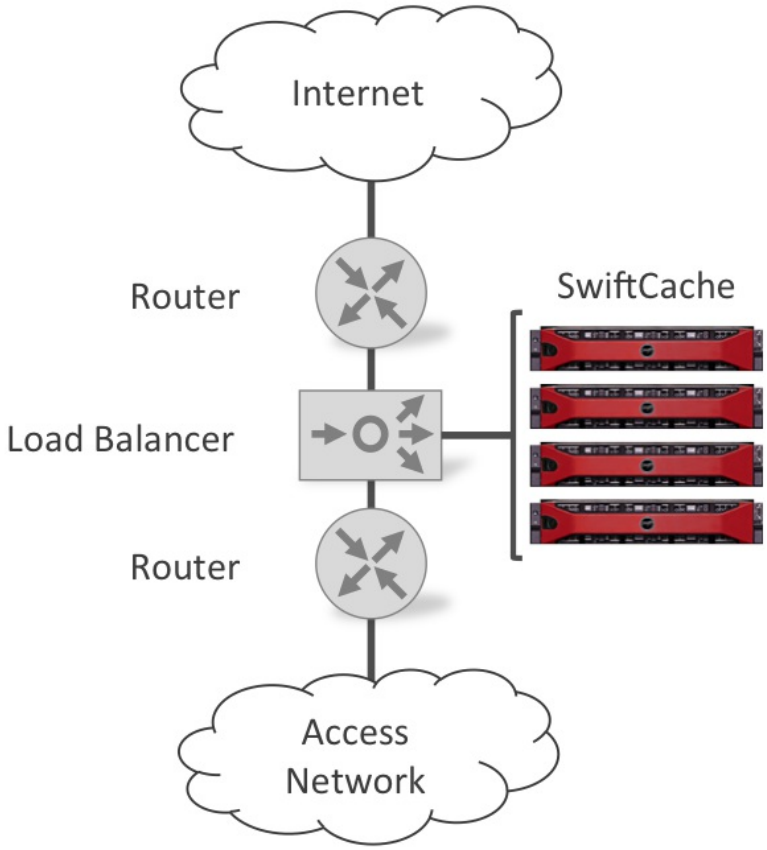
6.4.2 Out of Path/Router Mode



Out of Path/Route Mode uses *Policy Based Routing* (PBR) to redirect HTTP requests on port 80 to the SwiftCache. Only HTTP traffic is routed to the cache, so other traffic flows as it did before.

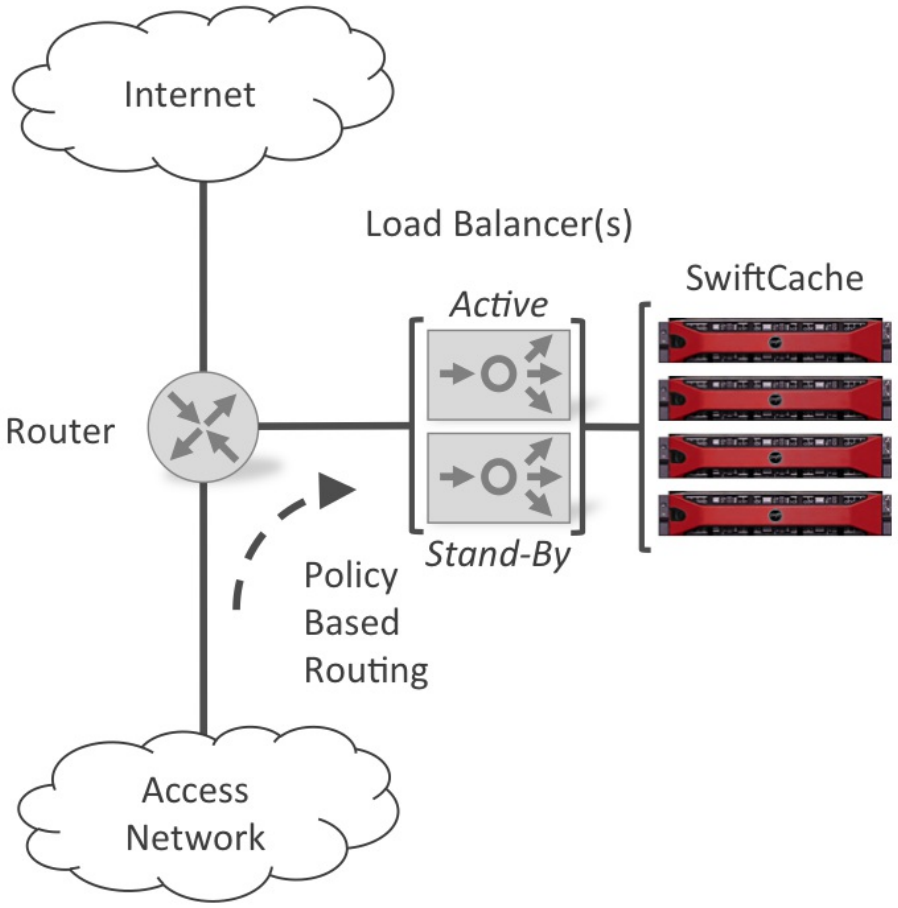
PBR provides flexible, fine-grained control over the traffic to intercept. It is straightforward to select a portion of users from the subscriber subnet to try out a new configuration before rolling out the changes to the entire subscriber base.

6.4.3 Load Balancer in Bridge Mode



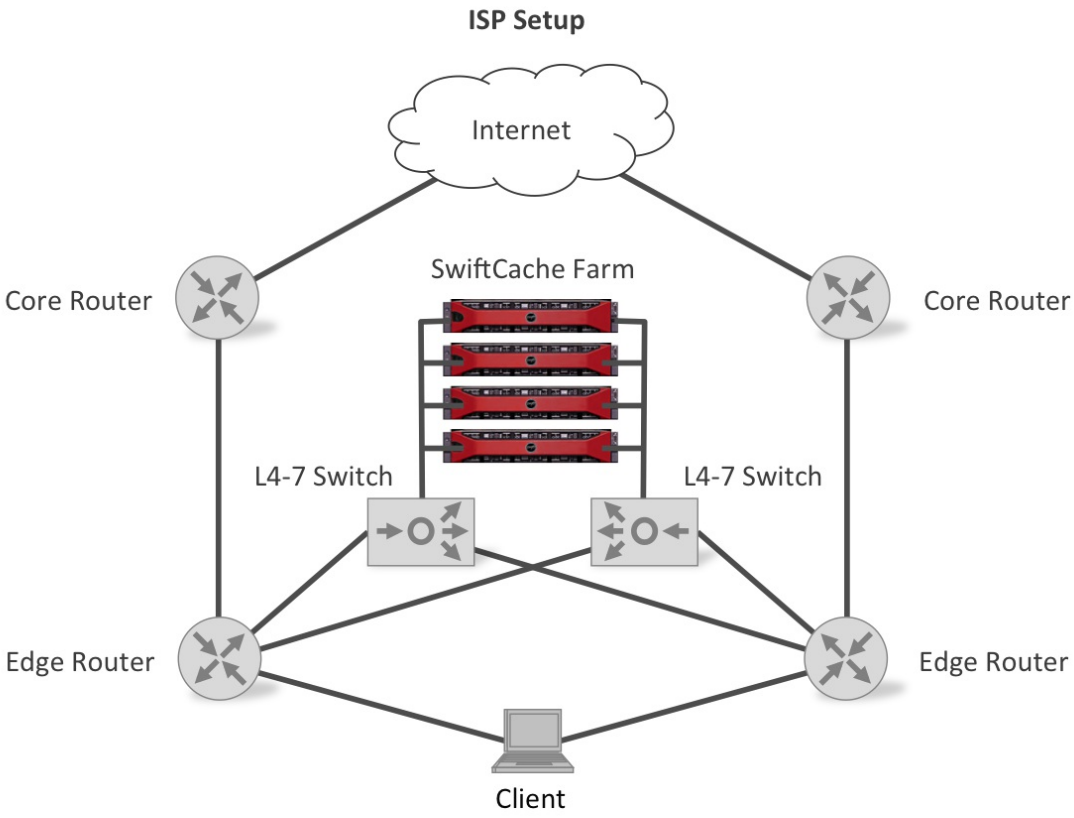
Using a load balancer in bridge mode does not require any *Policy Based Routing* (PBR). The load balancer is placed between two routers. The SwiftCache has its default route set to the router on the internet side of the load balancer. Static route(s) to the access network are added to the other one.

6.4.4 Load Balancer in Router Mode



This is the most common SwiftCache deployment topology. The routers have *Policy Based Routing* (similar to the *Out of Path* topology). In this case, the client HTTP connections are sent to the load balancer. For a high availability service, multiple load balancers can be used.

6.4.5 Internet Service Provider (ISP)



In a large ISP network with two international gateways, it is recommended to deploy SwiftCache with a cluster at each gateway. This simplifies the policy-based routing required to direct traffic to the SwiftCache cluster without introducing any asymmetry to the traffic flow.

The diagram above shows:

- the SwiftCaches are deployed in two clusters that are dual-homed to a pair of Layer 4-7 switches;
- each switch hangs off a leg to the edge router;
- the edge routers are configured with policy-based routing on the north and southbound interfaces to direct web traffic to each SwiftCache cluster appropriately.

Avoiding Asymmetric Routing

If there are local Content Delivery Network (CDN) servers in the network, then it is important to avoid creating asymmetric routing paths.

For example, a SwiftCache is operating in full-transparency mode and is spoofing the client's IP address. If it sends on the request to a local CDN server, the response would return directly from the CDN server back to the client, and not via the SwiftCache. This would break the traffic path and cause the connection to fail.

The request's network route out differs from the route back, so this is known as *asymmetric routing*.

To avoid asymmetric routing the SwiftCache can be configured to disable IP spoofing specifically when being used with local CDN servers. This means that the SwiftCache falls back to a semi-transparent mode of operation

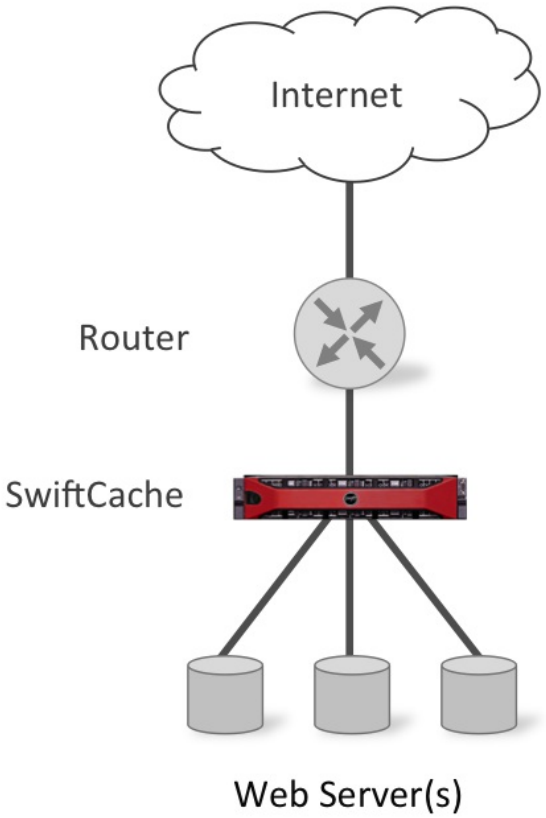
just for the sites served by the local CDN servers. This preserves the routing symmetry.

To disable IP spoofing just for local CDN servers a policy should be created that matches on the CDN server's IP address(es). Please see the **Policy** section later in this manual for more information about configuring policies.

For example, if the local CDN servers are deployed in the netblock `10.0.0.0/24`:

```
[policy-local_CDN]
match server subnet 10.0.0.0/24
ip_spoofing = off
```

6.4.6 Reverse Proxy



SwiftCache can be deployed in a reverse proxy arrangement in order to handle large traffic loads on behalf of the origin servers. This might be for use internally within an office intranet or with public-facing servers on the Internet.

Offloading the traffic throughput to SwiftCache in this manner means that the origin servers can be lighter in hardware specification, reduced in number or even replaced with consolidated virtual machines.

6.5 Practical Deployment Considerations

6.5.1 Physical Connections

The SwiftCache appliance may require one or more network connections depending on the configuration options chosen. Typically there would be two gigabit Ethernet ports bonded into one virtual interface (with corresponding configuration on the network switch), or a single 10-gigabit Ethernet port.

6.5.2 Pilot and Rollout

As with any significant network infrastructure changes, it is important to avoid disruption to end-users. Typical large-scale installations are rolled out progressively.

During an initial pilot phase, a small subset of traffic and/or users are routed via SwiftCache (one or more appliances) to allow the SwiftCache configuration to be tuned for the best performance.

Once this is achieved, the traffic load should be gradually increased and the performance monitored for any opportunities to further fine-tune the configuration.

6.5.3 Sizing

It is recommended that SwiftCache is deployed in a *N+1 redundant* cluster so if any one appliance fails, the remainder can handle the traffic. N+1 redundancy means that for any given number of appliances (N), there is at least one independent appliance available as a backup (+1).

The smallest recommended cluster size is therefore two SwiftCache appliances. This arrangement also allows upgrades to be performed without any interruption to service.

6.5.4 Load Balancing

A load balancer is an integral part of a clustered SwiftCache deployment as it enables the solution to scale to very large volumes of traffic. The load balancer is responsible for directing requests across the SwiftCache cluster to keep the workload evenly distributed across all nodes.

An intelligent Layer 4-7 switch or load balancer will use an efficient hashing algorithm to ensure that client requests for same item are sent back to the same SwiftCache; this is important for maximum throughput.

Most load balancers will also employ application health monitors to ensure that the SwiftCache service is up and available to respond to requests. This is essential for a large SwiftCache deployment where a fully-transparent mode of operation is required.

Compatible Layer 4-7 Switches and Load Balancers

SwiftCache is compatible with any network switch, but has been tested with the following devices:

- Citrix Netscaler
- Cisco ACE
- F5 LTM
- A10
- Brocade ADX

Direct Connections from Load Balancers

Some load balancers, such as the Citrix Netscaler, create a direct connection to the SwiftCache on the proxy port. With load balancers of this type, it is necessary to create a specific policy on SwiftCache to allow the direct connection from the load balancer.

In most deployment scenarios direct connections from clients are disabled as it is not desirable for the SwiftCache to operate as an explicit proxy.

A sample policy is shown below. See the **Policies** chapter later in this manual for more information about configuring policies.

For example, the load balancer will forward connections from the subnet `10.0.0.0/16`:

```
[policy-loadbalancer]
match client subnet 10.0.0.0/16
allow_netscaler = on
```

7 SwiftCache User Interfaces

7.1 Overview

SwiftCache provides two different user interfaces for configuration and administration of the system:

- a web-based, graphical user interface (*GUI*) accessed via a web browser; and
- a command-line interface (*CLI*), accessed directly via a console login or serial port connection on the SwiftCache, or remotely via a Secure Shell (*SSH*) login session.

Certain advanced configuration options are only available from the CLI. Similarly, some aspects of monitoring and reporting are only available from the GUI.

In addition to these methods of configuration, an additional web service called *SwiftSense* is available. This aggregates alerts and reporting information from SwiftCache appliances and offers extended capabilities for business reporting. For more information on SwiftSense, please see the **SwiftSense** chapter later in this manual.

By default SwiftCache's web GUI runs on port 8500. It is not encrypted initially, but can be configured to use SSL encrypted should it be required. SwiftCache's SSH interface is on port 22. Direct console logins can be achieved by either logging in via a serial port connection, or using a keyboard and monitor directly connected to the SwiftCache appliance.

For more information on the GUI, please refer to the **SwiftCache GUI** chapter later in this manual. For more information on the CLI, please see the **SwiftCache CLI** chapter.

7.2 User Accounts

By default there are two user accounts that an operator can use to login to a SwiftCache. These are:

- the `admin` user, who may monitor the SwiftCache and has the ability to make changes to configuration options including shutdown and restart; and
- the `monitor` user, who can only see reporting information, statistics and alerts and cannot view or change the configuration of the SwiftCache appliance.

The `monitor` user can only connect to the SwiftCache using the GUI.

7.3 Configuration Daemon

A *daemon* is a software program that runs as a background process. Access to the configuration database is handled by a component of SwiftCache called `configd` (the configuration daemon). Both the CLI and GUI communicate with `configd` in order to read the configuration to display it to the operator and to update the configuration when a value is changed.

7.4 Configuration Keys

SwiftCache is configured using *configuration keys* and their corresponding values. These configuration keys define different settings depending on the value that they are given.

Configuration keys may expect a certain type of value. For example, if the configuration key expects an IP address as its value, only a IP address may be provided. Similarly, if a configuration key expects a numerical

value, a text string may not be provided. SwiftCache will check that configuration values are appropriate and will not permit the wrong type of value to be configured.

You can find a list of possible configuration keys and values at **Appendix C**.

7.4.1 Examples of Configuration Keys and Values

`cache_delay = 2`	The `cache_delay` option is set to a value of 2
`icc_enabled = no`	The `icc_enabled` option is disabled (i.e. set to `no`)
`allow_explicit = on`	The `allow_explicit` proxy mode option is enabled (i.e. set to `on`)

Common types expected for configuration values can be *Boolean*, *text string*, *number* and *IP address*.

Boolean values can be specified as `yes`, `on` and `true`, or as `no`, `off` and `false`.

8 SwiftCache GUI

SwiftCache's graphical user interface (GUI) provides access to the most frequently-used monitoring and configuration options of a SwiftCache appliance. It is accessed using a web browser.

8.1 GUI Access

The GUI runs on port 8500 of the SwiftCache appliance by default. It can be accessed remotely with a web browser either by hostname or IP address, which will be determined by your network setup. For example:

```
http://yourswiftcache.example.org:8500/
```

where `yourswiftcache.example.org` is replaced with the hostname or IP address of the actual appliance.

It is recommended that SSL encryption is enabled for all communications with the GUI, and that the GUI is protected from the public Internet and unauthorised access by a firewall or other appropriate network security. For more detail on how to secure the SwiftCache GUI, please refer to the **Securing SwiftCache** section later in this manual.

If you have configured SSL encryption for the SwiftCache GUI then please note that you will need to connect using `https:` in your web browser rather than `http:` in order to access the GUI. For example:

```
https://yourswiftcache.example.org:8500/
```

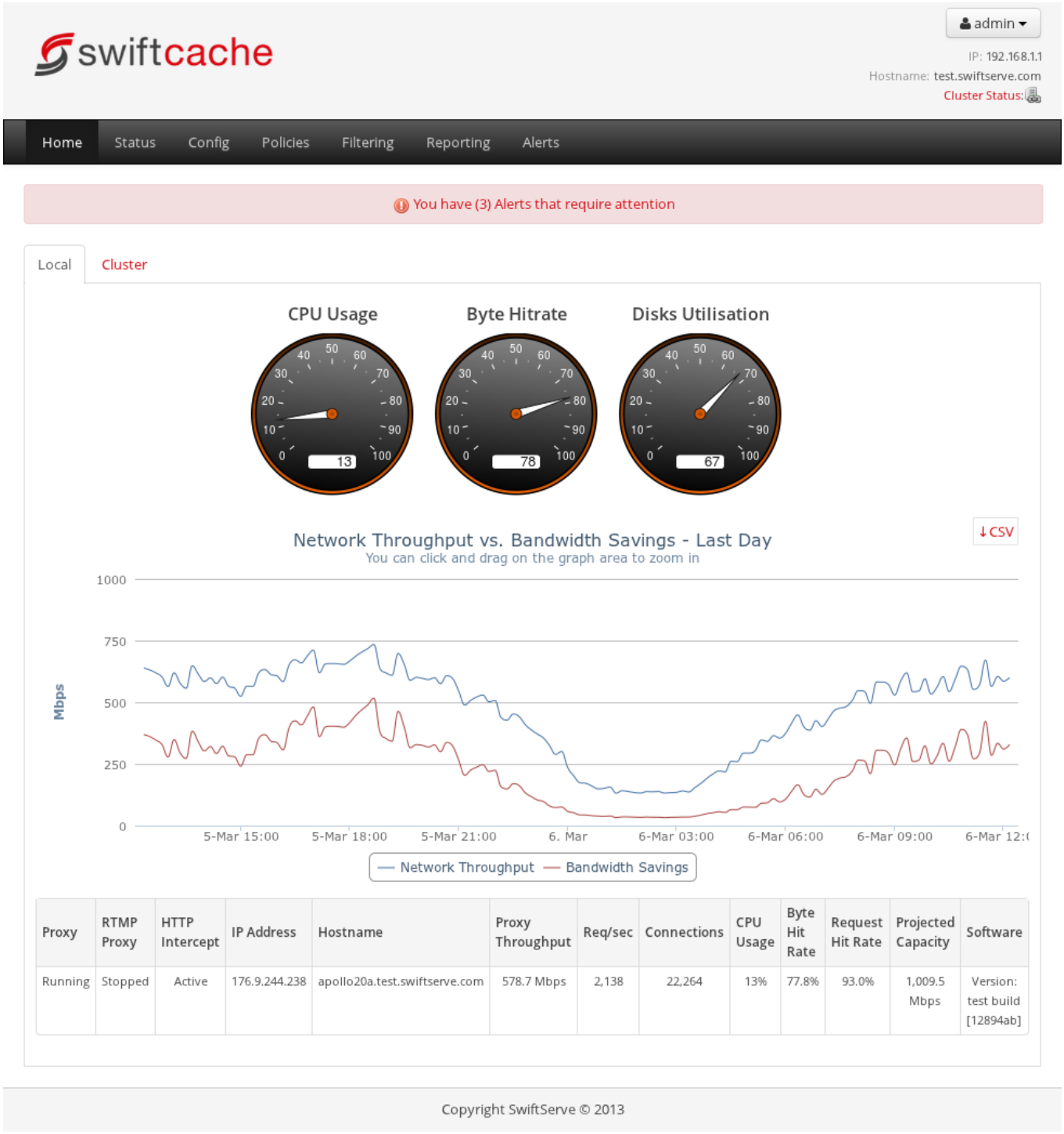
where `yourswiftcache.example.org` is replaced with the hostname or IP address of the actual appliance.

For the purposes of this manual, subsequent URL examples for the GUI will be given using `http:` only.

8.2 Logging In

When logging in to the GUI for the first time, enter the username `admin` and the password supplied by your vendor.

Once you have logged into the SwiftCache web interface for the first time you will be presented with the main dashboard of the appliance. From here you can see if there are any alerts that have been generated, the current utilisation and the network statistics for the last twenty-four hours.



8.2.1 SSL Warnings on GUI Login

If the GUI is configured to use SSL, the web browser may display a warning about the site's security certificate. This is because SwiftCache uses a self-signed SSL certificate. A self-signed certificate still allows the connection to the GUI to be encrypted securely, however it has not been signed by a certificate authority recognised by the web browser.

In this specific case the warning can be safely ignored by performing one of the following actions depending on the web browser used:

- click on *"Continue to this website"* (Internet Explorer);
- click on *"I understand the risks"* and create an exception, or *"Accept this certificate permanently"* (Mozilla Firefox);

- click on *"Continue"* (Safari);
- click on *"Proceed anyway"* (Chrome).

For other web browsers, please consult their documentation.

8.3 Finding Help in the GUI

8.3.1 Contextual Help

The SwiftCache GUI has a built-in contextual help system that provides information for all configuration keys and pop-up hints, or *tooltips*, for certain features and functions.

Tooltip Example

Hovering your mouse pointer over any configuration key item name will cause a tooltip to be displayed that contains help information relevant for that setting.

8.4 Sections of the GUI

The SwiftCache GUI is divided into tabs corresponding to each section below.

8.4.1 Home Tab (Dashboard)

The *Home* tab provides SwiftCache operators with a quick overview of the status, health and performance of the appliance known as the *dashboard*. It displays any alerts that have been generated, the current appliance utilisation and its network statistics for the last twenty-four hours.

If the appliance is part of a cluster then there will be both a *Local* tab and a *Cluster* tab: the Local tab provides information about this appliance; the cluster tab shows information about all the nodes in the cluster.

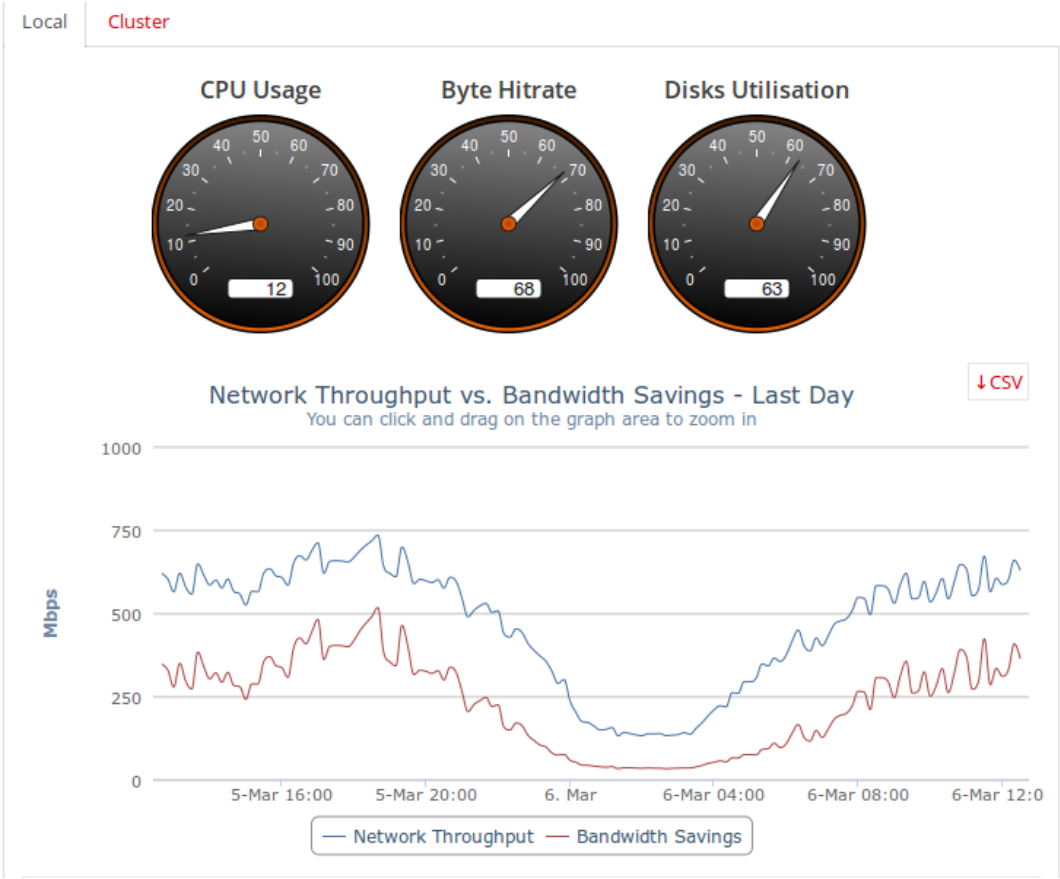
The dashboard is broken into three major components: an alert summary, performance information and the status table.

Alert Summary

The first section of the dashboard provides a count of unacknowledged alerts, which can be expanded to display summaries of each type of alert.

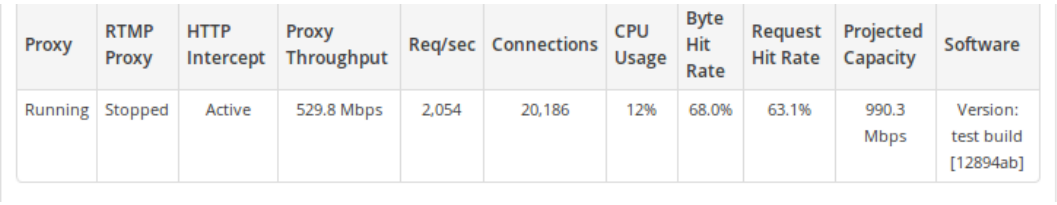
The alert summary is available in the header of all pages within the GUI for easy access to unacknowledged alerts.

Performance Information



The next section of the dashboard shows performance information including CPU usage and disk utilisation, as well as network throughput versus bandwidth savings from the last twenty-four hours.

Status Table



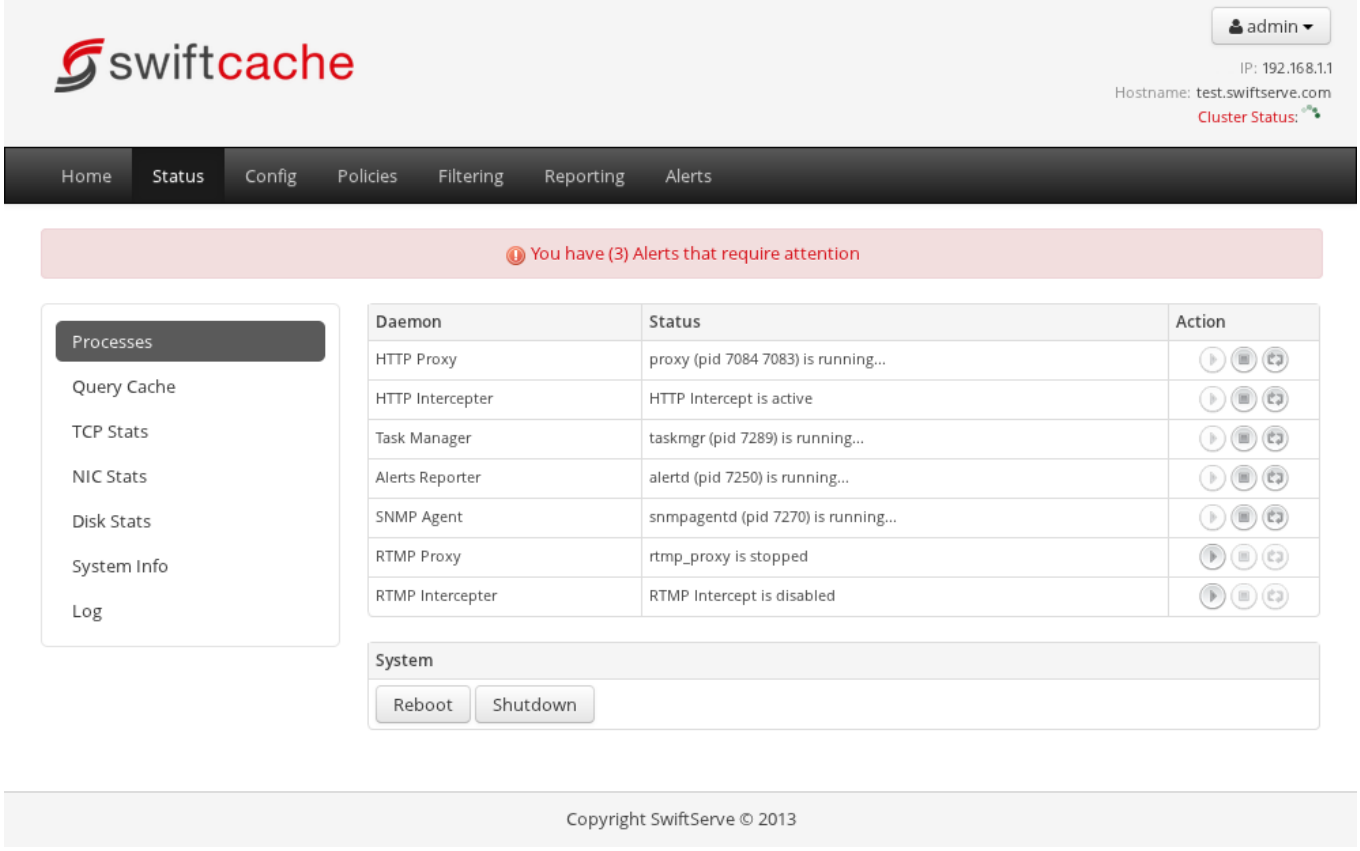
Proxy	RTMP Proxy	HTTP Intercept	Proxy Throughput	Req/sec	Connections	CPU Usage	Byte Hit Rate	Request Hit Rate	Projected Capacity	Software
Running	Stopped	Active	529.8 Mbps	2,054	20,186	12%	68.0%	63.1%	990.3 Mbps	Version: test build [12894ab]

This last section of the dashboard provides a table of more detailed status breakdown for each appliance within the cluster. This information includes the proxy and intercept status, IP address, hostname, throughput, health, hit rates, projected capacity and software versions.

8.4.2 Status Tab

The *Status* tab provides a more detailed overview of the current performance and status of the SwiftCache appliance. It includes a number of low-level networking and operating system statistics that are useful for troubleshooting. It is divided into subsections.

Processes



The *Processes* subsection shows information the about the various proxy processes (or *daemons*). There are several processes that provide different services and capabilities on a SwiftCache appliance.

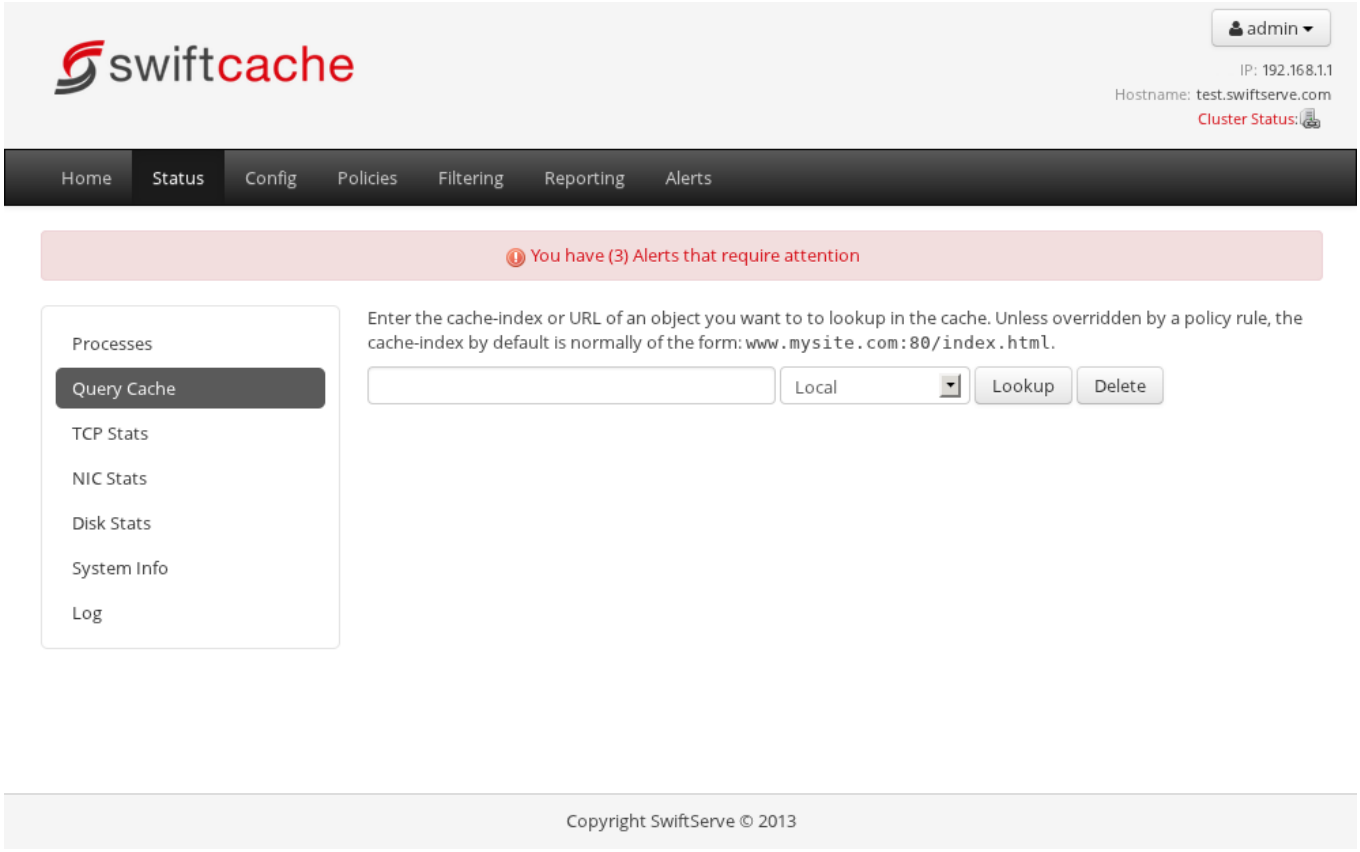
Proxy	The main SwiftCache proxy process that handles HTTP traffic.
Intercepter	Intercepts HTTP traffic on port 80 and passes it to the SwiftCache proxy process.
Task Manager	Handles log rotation and other admin processes.
Alerts Reporter	Handles creation and sending of alerts.
SNMP Agent	Provides SNMP information to external monitoring systems.
RTMP Proxy	SwiftCache RTMP proxy process for handling RTMP traffic.
RTMP Intercepter	Intercepts RTMP traffic on port 1935 and passes it to the RTMP proxy process.

The table shows current status of each daemon and provides buttons to start, stop or restart each process individually. The RTMP Proxy and Interpreter processes will be shown greyed-out if the RTMP option is not enabled in the current license key.

Underneath there are buttons to reboot and shutdown the appliance.

Care should be taken to avoid causing any disruption to client connections to the SwiftCache appliance. Remove traffic from the SwiftCache before stopping any processes or initiating an appliance reboot or shutdown.

Query Cache



The *Query Cache* subsection allows the operator to determine if a particular object is known to the proxy, and if so, some information about the way it is stored and has been accessed. This page also allows operators to purge objects from the cache manually if they specifically need to be made unavailable. In normal operation, old items are automatically purged from the cache over time.

The search box allows operators to query the cache to view the status of an object. Queries are performed by entering the URL of an object or its cache index. The response will display the following details for objects cached both normally and compressed (*gzip*):

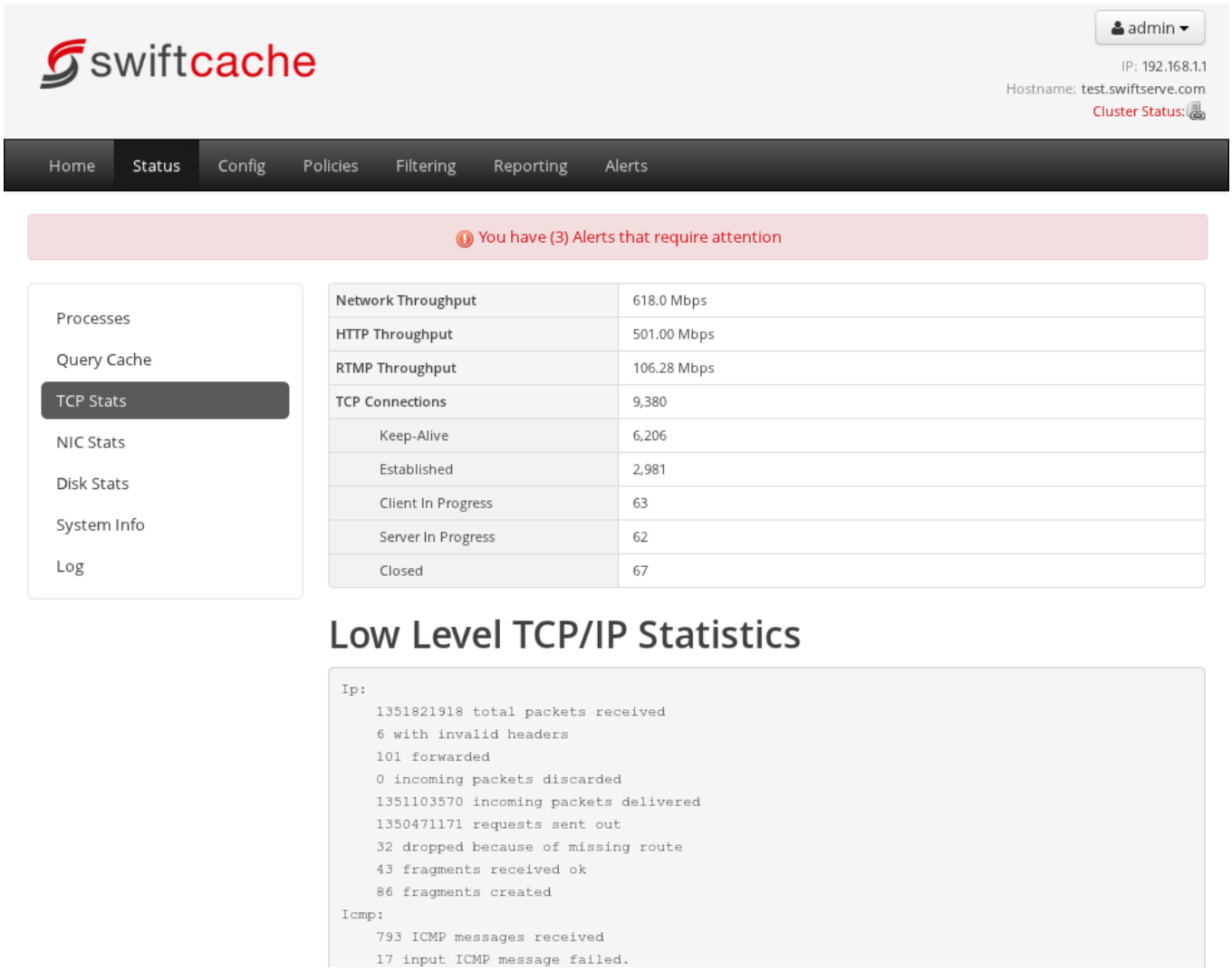
Index	The unique cache identifier for the object.
Content-Length	The size in bytes of the object.
Last-Validated	The last date and time that the cache object was verified to be correct (compared to the origin server)
Created	The date and time when the cache object was created.
Last-Accessed	The last date and time when the object was accessed by a client.
Expires	The date and time when the cache object will be deemed to no longer be valid and will be removed from the cache.
Server	The identifier of the origin server.
Last-Modified	The date and time when the original file was created.
ETag	The hash of the object.
Location	Whether the object is stored in memory or on disk.
Path	The location of the object (i.e. on which disk).
Content-Type	The MIME type of the object.

A cache index is generated by removing the protocol specifier (e.g. `http://`) and adding the port to the request URL, for example:

```
www.mysite.com:80/index.html
```

Please note however that the format of a cache index may be modified through the use of the `cache_index` policy key.

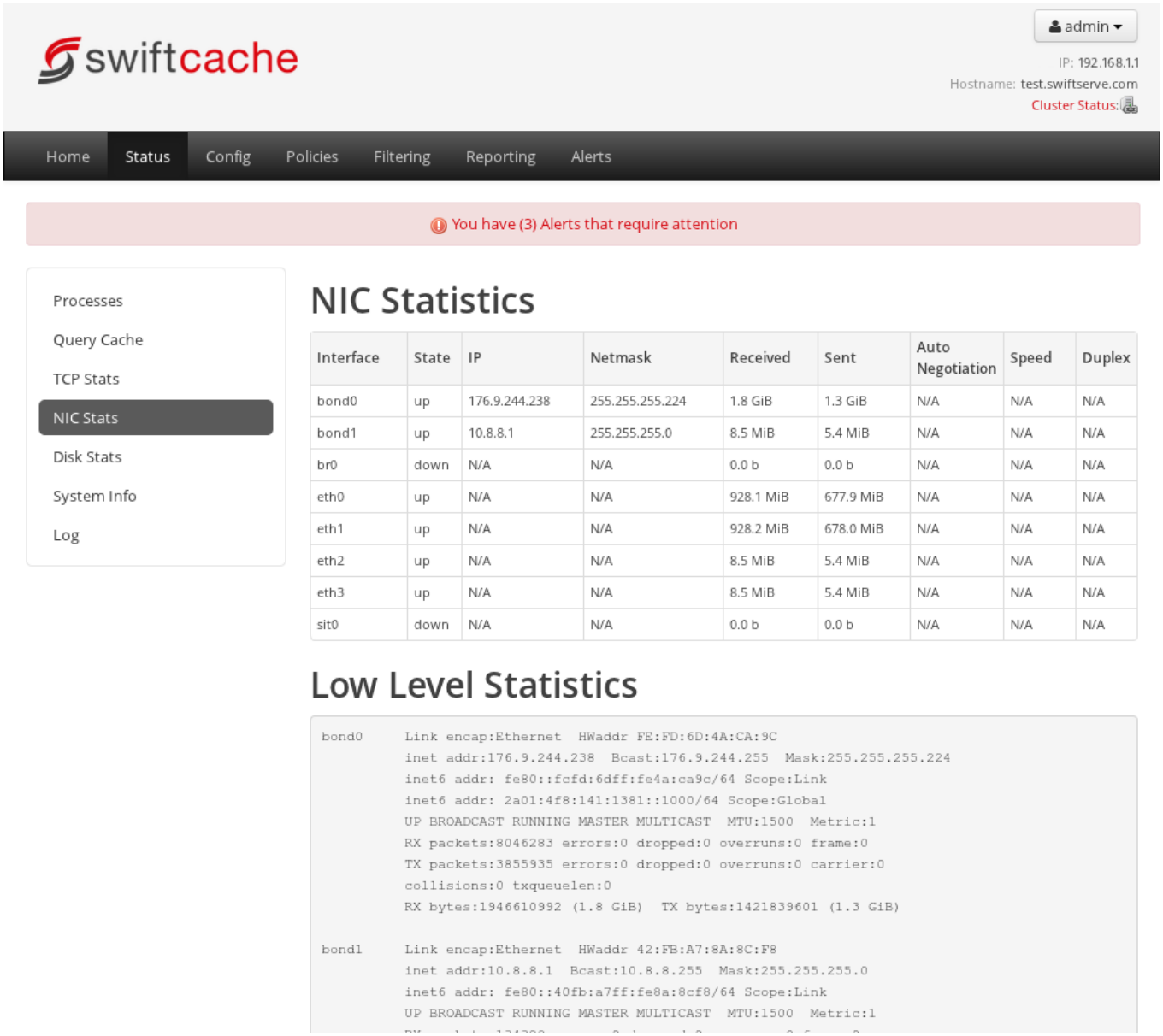
TCP Stats



The *TCP Stats* subsection shows the current throughput of the SwiftCache appliance. Separate figures are displayed for network, HTTP and RTMP throughput, the total number of TCP connections and the number of connections in each state (*Keep-Alive*, *Established*, *Client In Progress*, *Server In Progress* and *Closed*).

Underneath, some more detailed networking statistics of the SwiftCache are provided. This portion of the report can be particularly useful to help experienced networking technicians to identify errors in the TCP/IP stack.

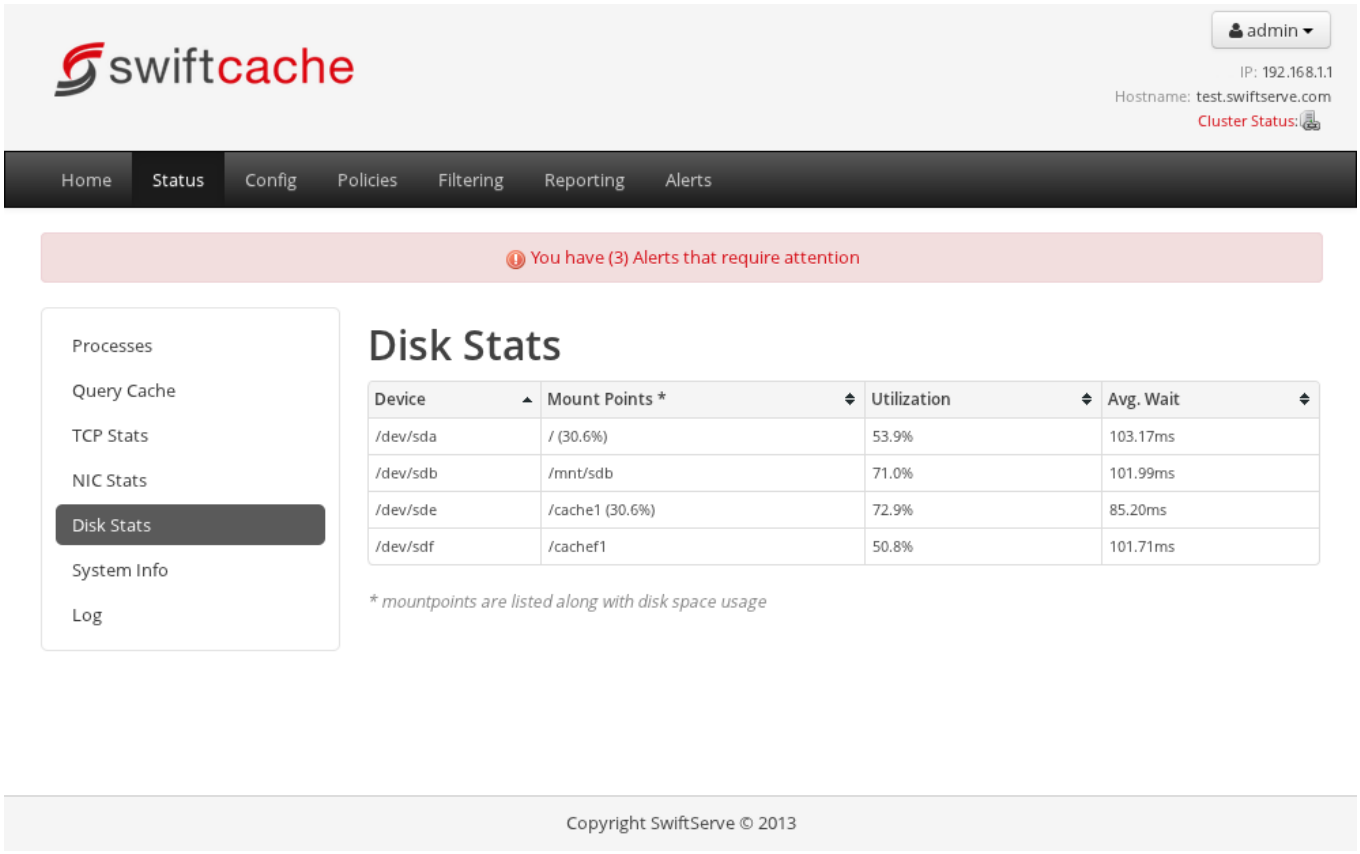
NIC Stats


swiftcache
admin
IP: 192.168.1.1
Hostname: test.swiftserve.com
Cluster Status:
Home
Status
Config
Policies
Filtering
Reporting
Alerts
You have (3) Alerts that require attention
Processes
Query Cache
TCP Stats
NIC Stats
Disk Stats
System Info
Log
NIC Statistics
Interface State IP Netmask Received Sent Auto Negotiation Speed Duplex
bond0 up 176.9.244.238 255.255.255.224 1.8 GiB 1.3 GiB N/A N/A N/A
bond1 up 10.8.8.1 255.255.255.0 8.5 MiB 5.4 MiB N/A N/A N/A
br0 down N/A N/A 0.0 b 0.0 b N/A N/A N/A
eth0 up N/A N/A 928.1 MiB 677.9 MiB N/A N/A N/A
eth1 up N/A N/A 928.2 MiB 678.0 MiB N/A N/A N/A
eth2 up N/A N/A 8.5 MiB 5.4 MiB N/A N/A N/A
eth3 up N/A N/A 8.5 MiB 5.4 MiB N/A N/A N/A
sit0 down N/A N/A 0.0 b 0.0 b N/A N/A N/A
Low Level Statistics
bond0 Link encap:Ethernet HWaddr FE:FD:6D:4A:CA:9C
inet addr:176.9.244.238 Bcast:176.9.244.255 Mask:255.255.255.224
inet6 addr: fe80::fcfd:6dff:fe4a:ca9c/64 Scope:Link
inet6 addr: 2a01:4f8:141:1381::1000/64 Scope:Global
UP BROADCAST RUNNING MASTER MULTICAST MTU:1500 Metric:1
RX packets:8046283 errors:0 dropped:0 overruns:0 frame:0
TX packets:3855935 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:1946610992 (1.8 GiB) TX bytes:1421839601 (1.3 GiB)
bond1 Link encap:Ethernet HWaddr 42:FB:A7:8A:8C:F8
inet addr:10.8.8.1 Bcast:10.8.8.255 Mask:255.255.255.0
inet6 addr: fe80::40fb:a7ff:fe8a:8cf8/64 Scope:Link
UP BROADCAST RUNNING MASTER MULTICAST MTU:1500 Metric:1

The *NIC Stats* subsection shows the state, configured IP address and netmask of each network interface. Other information for each interface is displayed, including the current auto-negotiation state, speed and duplex settings, and also the total amount of data transmitted and received since the last reboot.

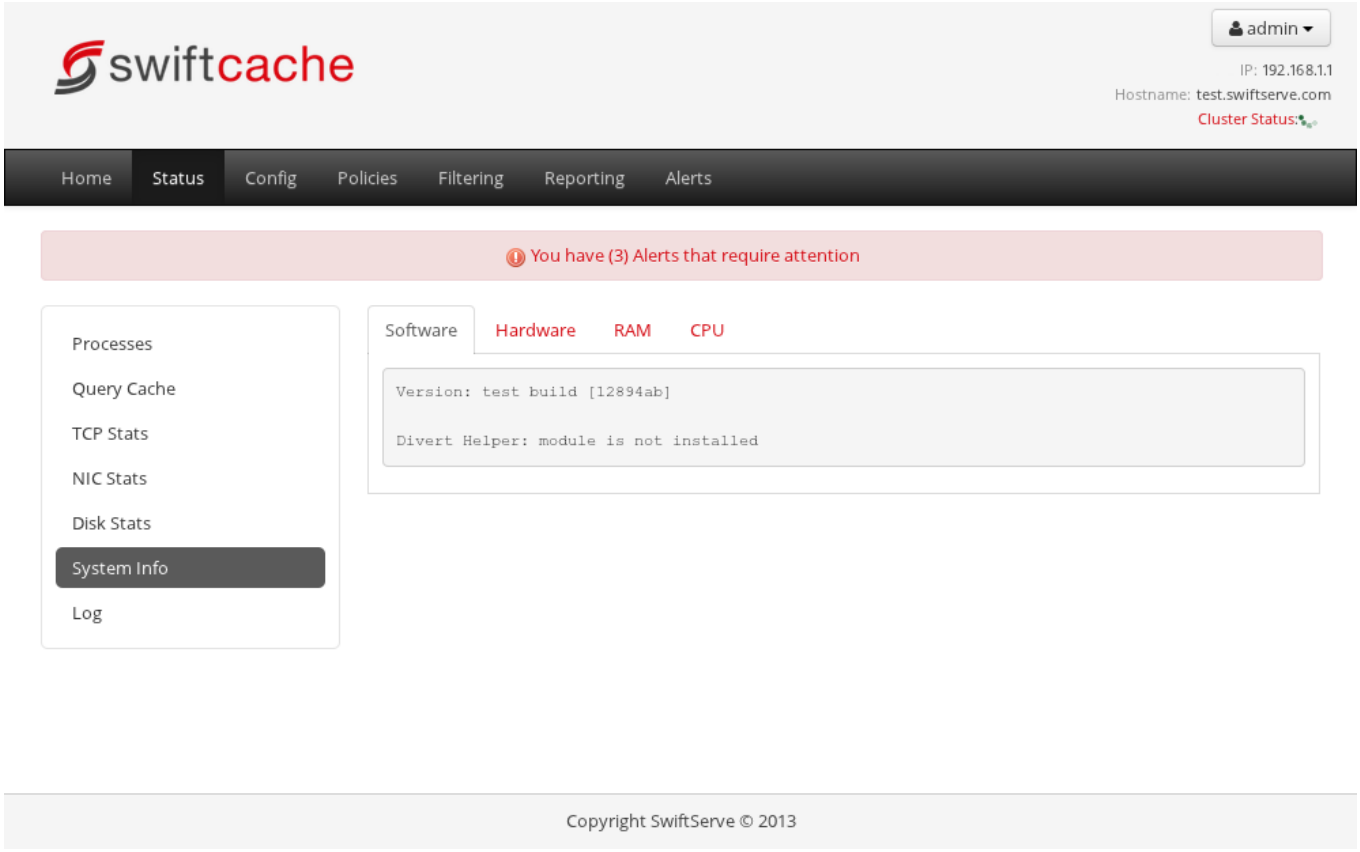
Underneath some more detailed interface statistics are provided. This portion of the report can be particularly useful to help experienced networking technicians to identify errors in the network.

Disk Stats



The *Disk Stats* subsection shows information on all the hard disks in the appliance: the device nodes, mount points, usage and utilisation are shown along with the wait time in milliseconds. This can be used to determine whether each disk is performing normally or is reaching the limit of its performance.

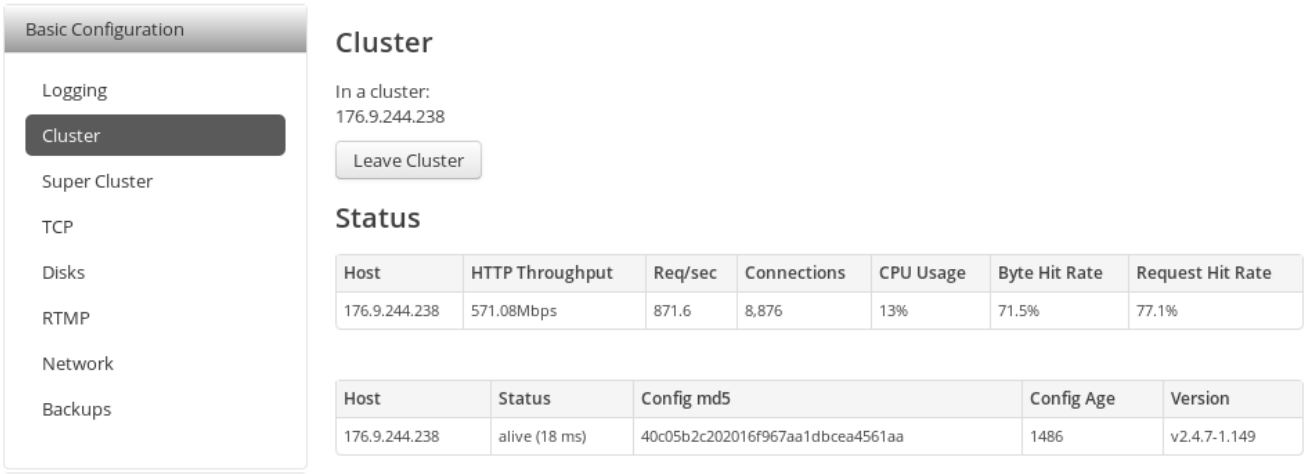
System Info



The *System Info* subsection contains system information about the SwiftCache appliance. The sub-tabs contain specific detail on the software, hardware, RAM and CPU.

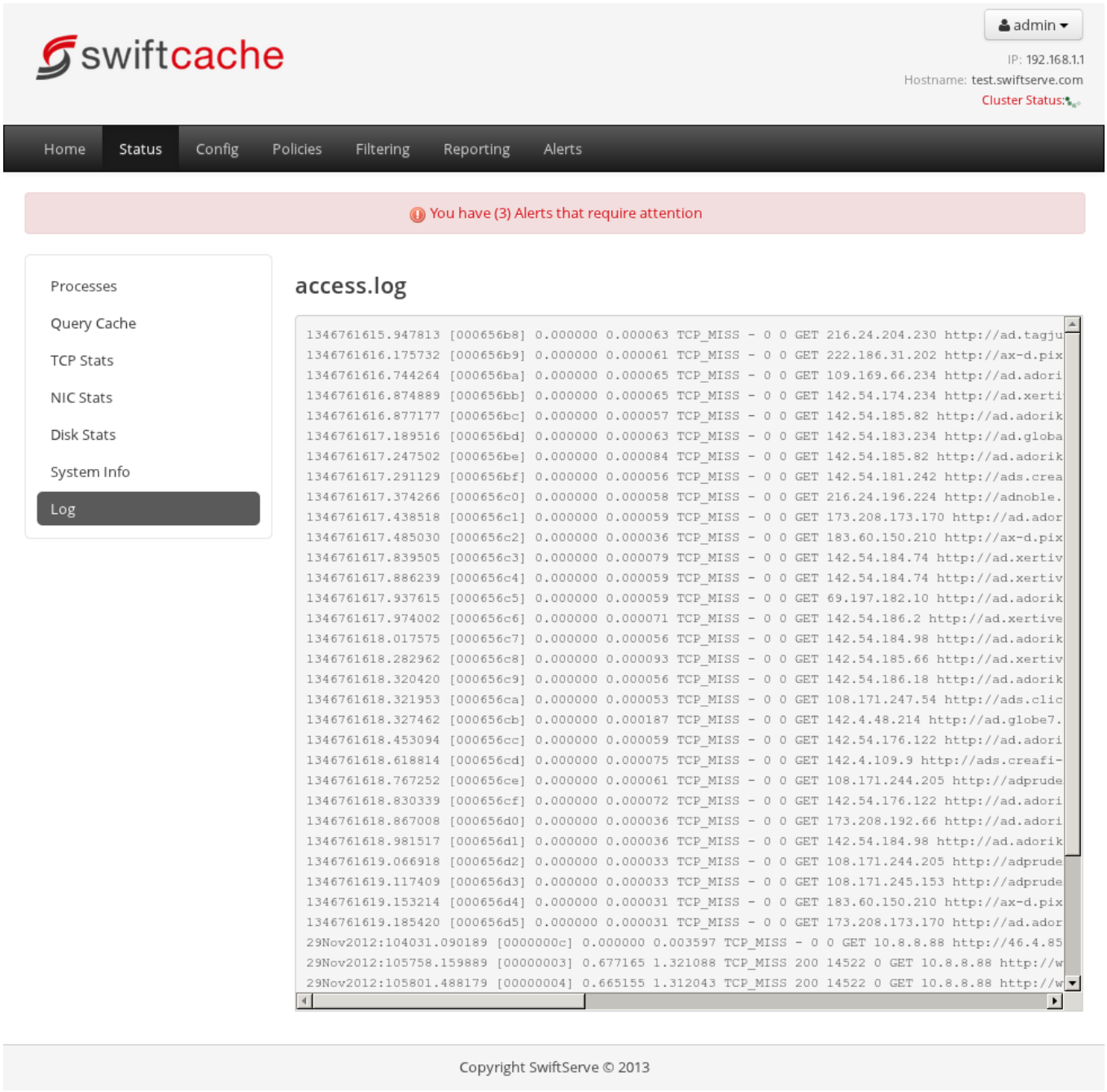
Software	Displays the SwiftCache software version. This can be used to determine if the appliance is running the desired/tested version of the software.
Hardware	Displays device information of the hardware components of the appliance. This can be used (for example) to determine the make and model of network interface card in case of problems with the connected switching hardware and/or kernel driver module version.
RAM	Displays free and total memory. This can be used to determine if there is any abnormality in the memory usage of the appliance.
CPU	Displays the CPU load and process table for the appliance. This can be used to check the overall load of the appliance and to see if any individual process has malfunctioned (note that it is normal for the *proxy* process to consume the majority of the CPU under high load conditions.

Cluster



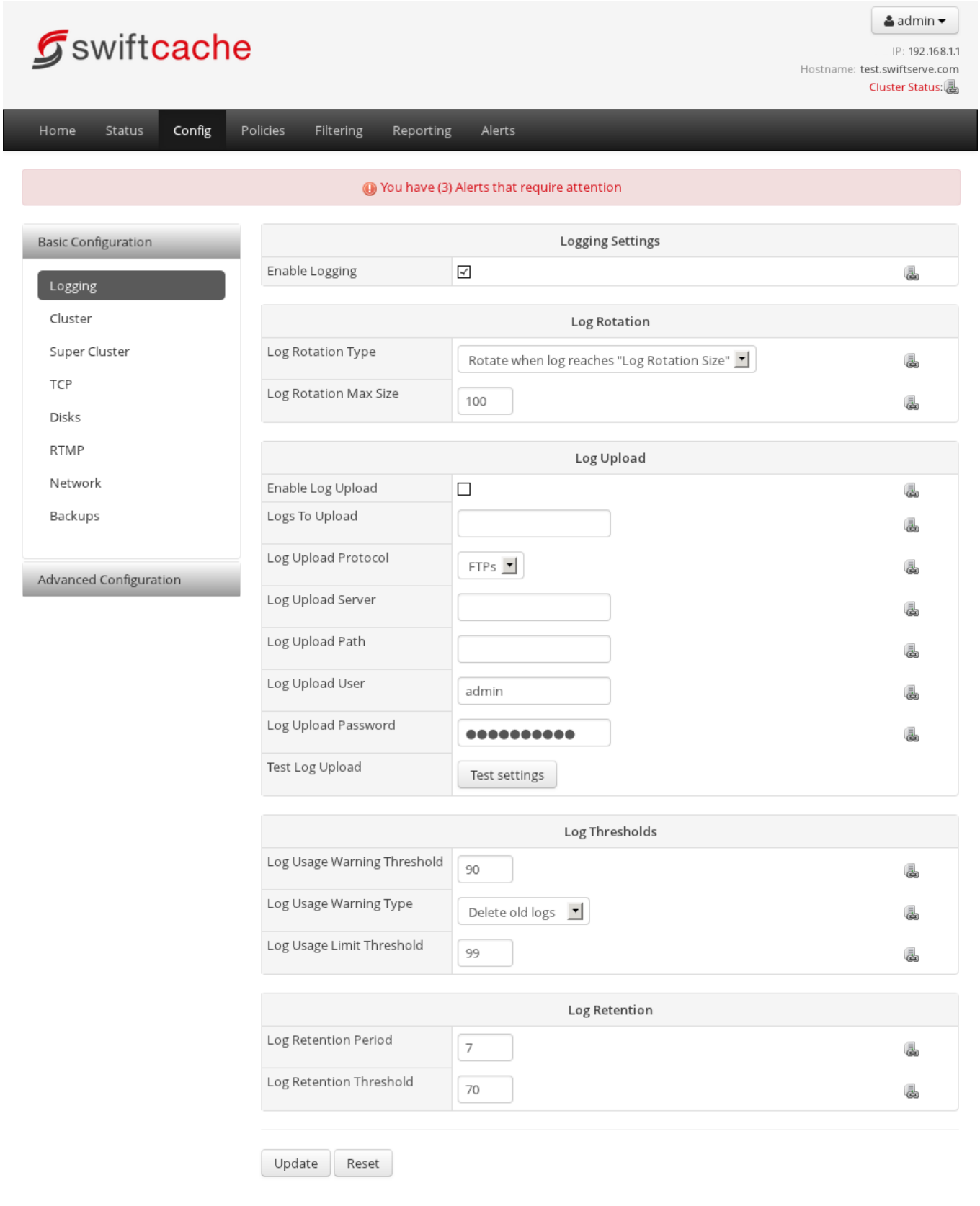
The *Cluster* subsection provides summary information for all of the machines within the cluster.

Log



The *Log* subection shows the most recent portion of the proxy access log as a quick check for operators. For more detailed problem diagnostics it is recommended to view the full log file directly.

8.4.3 Config Tab



The *Config* tab allows an operator to view and change the most common settings within the SwiftCache configuration. Policies, filtering, reporting and alerts are configured separately on their own tabs.

The navigation menu on the left of the page allows access to the different aspects of SwiftCache configuration,

split between basic and advanced configuration settings.

Changing Settings

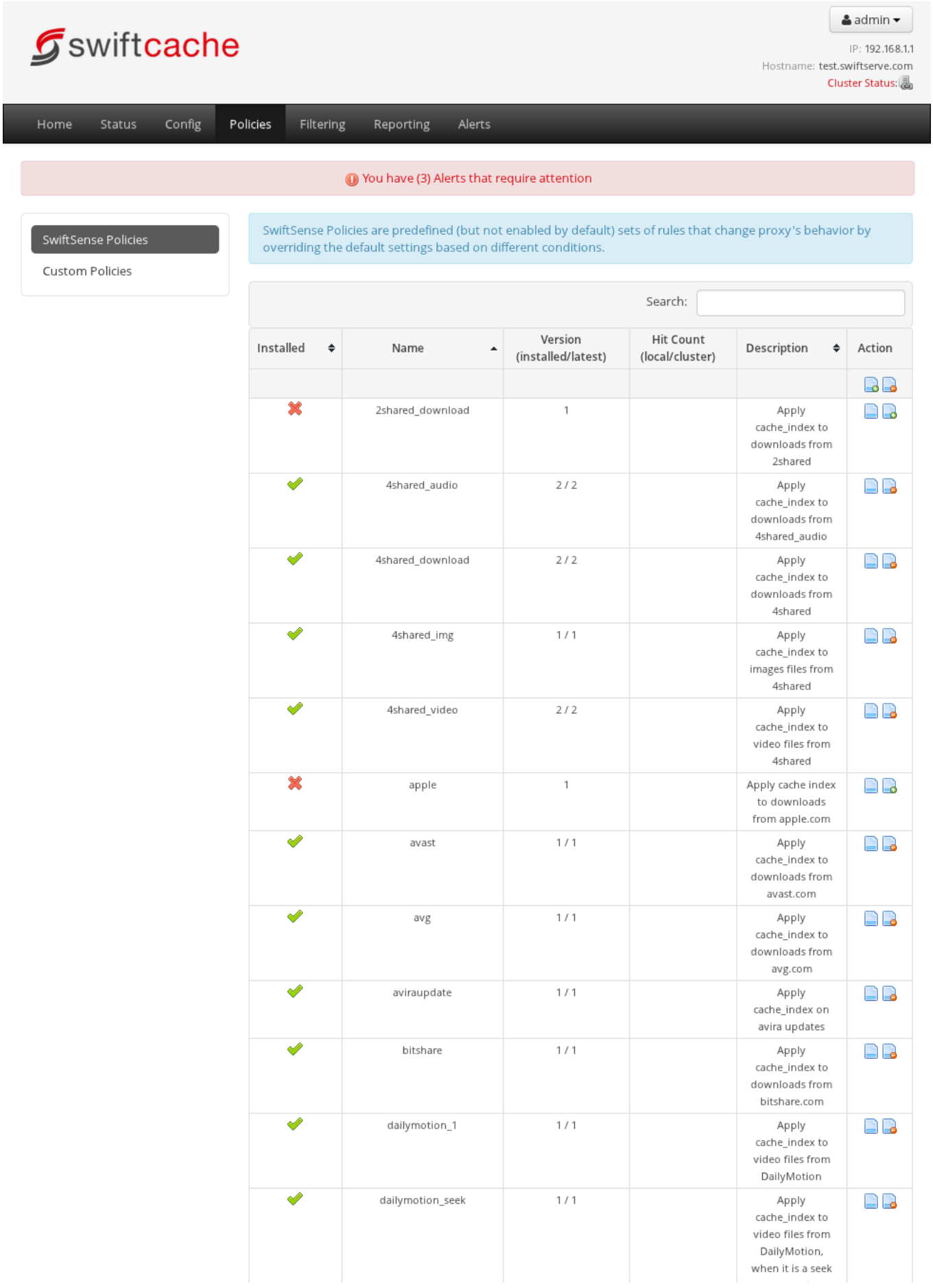
Settings are changed by updating the values using the text input fields, checkboxes or drop-down selection menus provided. Once a setting has been changed, click the *Update* button at the bottom of the page to confirm and apply the change. Alternatively, click the *Reset* button to revert all changes made on the page back to the currently-applied settings.

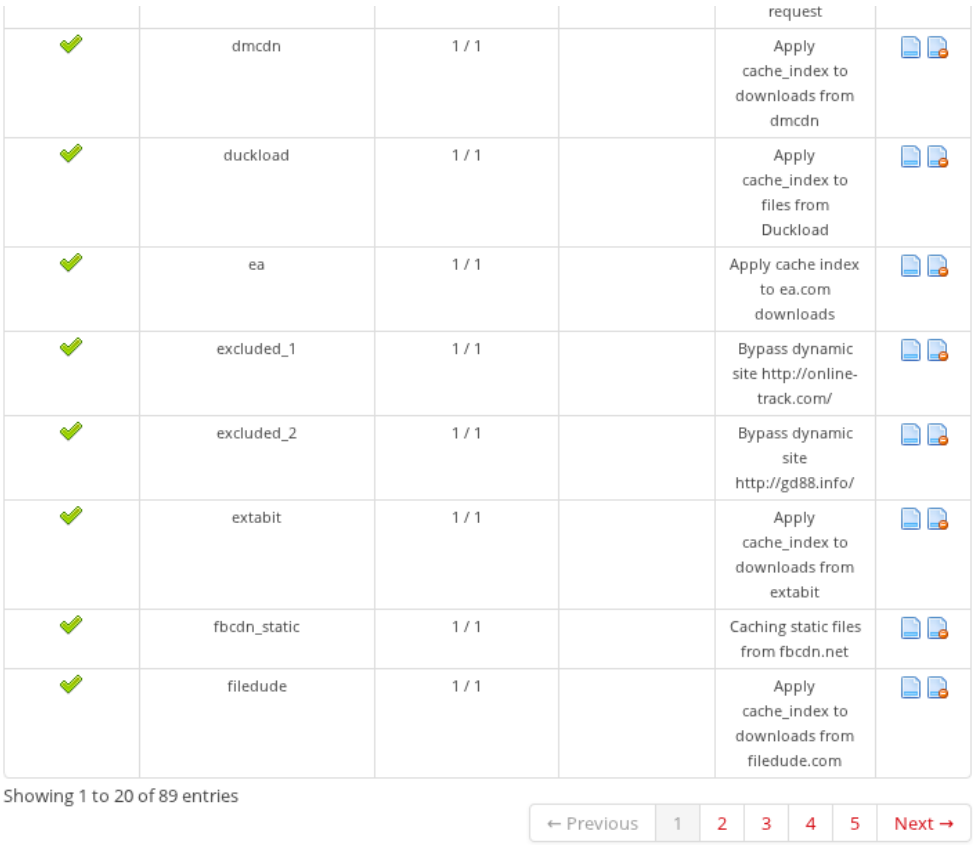
Care should be taken as changes are applied on clicking the *Update* button with no further user confirmation.

It is possible to change multiple settings on a single page at a time; simply make all the changes required and then click on the *Update* button to apply them.

Operators should apply all changes on a page if desired by clicking the *Update* button before browsing to other tabs, subsections or pages to avoid configuration changes being lost.

8.4.4 Policies Tab


swiftcache
admin
IP: 192.168.1.1
Hostname: test.swiftserve.com
Cluster Status:
Home
Status
Config
Policies
Filtering
Reporting
Alerts
You have (3) Alerts that require attention
SwiftSense Policies
Custom Policies
SwiftSense Policies are predefined (but not enabled by default) sets of rules that change proxy's behavior by overriding the default settings based on different conditions.
Search:
Installed
Name
Version (installed/latest)
Hit Count (local/cluster)
Description
Action
2shared_download 1 Apply cache_index to downloads from 2shared
4shared_audio 2 / 2 Apply cache_index to downloads from 4shared_audio
4shared_download 2 / 2 Apply cache_index to downloads from 4shared
4shared_img 1 / 1 Apply cache_index to images files from 4shared
4shared_video 2 / 2 Apply cache_index to video files from 4shared
apple 1 Apply cache index to downloads from apple.com
avast 1 / 1 Apply cache_index to downloads from avast.com
avg 1 / 1 Apply cache_index to downloads from avg.com
aviraupdate 1 / 1 Apply cache_index on avira updates
bitshare 1 / 1 Apply cache_index to downloads from bitshare.com
dailymotion_1 1 / 1 Apply cache_index to video files from DailyMotion
dailymotion_seek 1 / 1 Apply cache_index to video files from DailyMotion, when it is a seek



				request	
✔	dmcdn	1 / 1		Apply cache_index to downloads from dmcdn	
✔	duckload	1 / 1		Apply cache_index to files from Duckload	
✔	ea	1 / 1		Apply cache index to ea.com downloads	
✔	excluded_1	1 / 1		Bypass dynamic site http://online-track.com/	
✔	excluded_2	1 / 1		Bypass dynamic site http://gd88.info/	
✔	extabit	1 / 1		Apply cache_index to downloads from extabit	
✔	fbcdn_static	1 / 1		Caching static files from fbcdn.net	
✔	filedude	1 / 1		Apply cache_index to downloads from filedude.com	

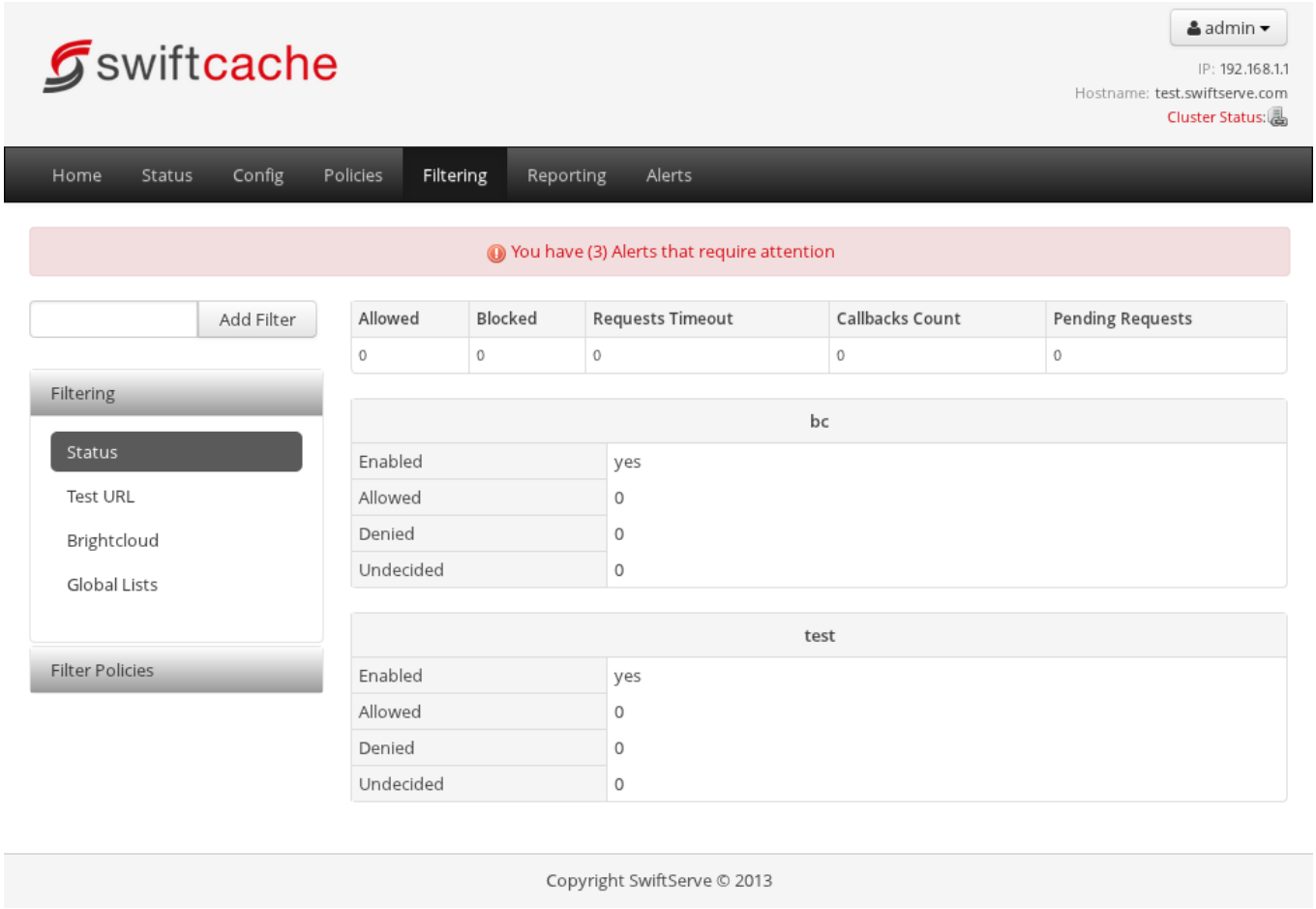
The *Policies* tab is used to manage the configuration of policies within SwiftCache. SwiftCache uses policies to define different cache behaviours for certain types of connection and for connections to specific sites.

Operators can apply different policies for different clients, including filter policies (defined under the *Filtering* tab), or change the proxy behaviour to improve the cache hit ratio on sites that are difficult to cache. This could be done, for example, by overriding cache behaviour defined by the origin server or client or by customising the SwiftCache's cache index.

There are two subsections, *SwiftSense Policies* and *Custom Policies*.

Please refer to the **Policies** section of this manual for more detailed configuration information.

8.4.5 Filtering



The *Filtering* tab allows operators to define a set of sites that are either blocked or allowed. The highest level of filtering is the global white and black lists that are configured on this tab. It is also possible to use dynamic URL classification where decisions are made on a site depending on the content, as identified by Brightcloud.

Filter Policies are also configured here, but for them to take effect it is also necessary to create one or more policies (defined under the *Policies* tab) that will apply them.

The navigation menu on the left of the page allows access to the different aspects of Filtering configuration, with two subsections, *Filtering* and *Filter Policies*. Above these is an *Add Filter* button which allows operators to add a new filter policy.

Please refer to the **Filters** section of this manual for more detailed configuration information.

The *Filtering* subsection has four pages, *Status*, *Test URL*, *Brightcloud* and *Global Lists*.

Status

This page shows the current total filter statistics in terms of the requests allowed and blocked, and underneath some statistics for the individual filter policies.

Test URL

This page allows an operator to enter a URL and validate it against the full filtering chain in order to determine what action would be taken. Information is provided about all types of filtering.

Brightcloud

This page allows the operator to see if the Brightcloud module is active and, if so, to check the category associated with a hostname (domain). The Brightcloud filtering module will be enabled automatically if the SwiftCache license includes it. For more information on Brightcloud, please refer to the **Brightcloud (Dynamic**

URL Classification) section in the **Filtering** chapter in this manual.

Global Lists

This page allows the operator to define the global black and white lists. The *Location* can be specified as an HTTP URL or a local path (to a file on the appliance). The *Refresh Period* defines how frequently it is updated and the *Redirect Location* indicates where denied requests are forwarded to if applicable.

8.4.6 Reporting

The *Reporting* tab shows graphs and statistics based on the historical performance of the SwiftCache appliance and cluster, which are continuously updated in real-time.

SwiftCache will aggregate and display statistics from across the cluster as well as from the local device. Selection of the report *scope* is enabled through tab controls:

Reports are divided in to subsections corresponding to *Traffic*, *Top 100*, *System*, *Performance*, and *SwiftSense*. These are described in more detail below.

Interacting with Graphs

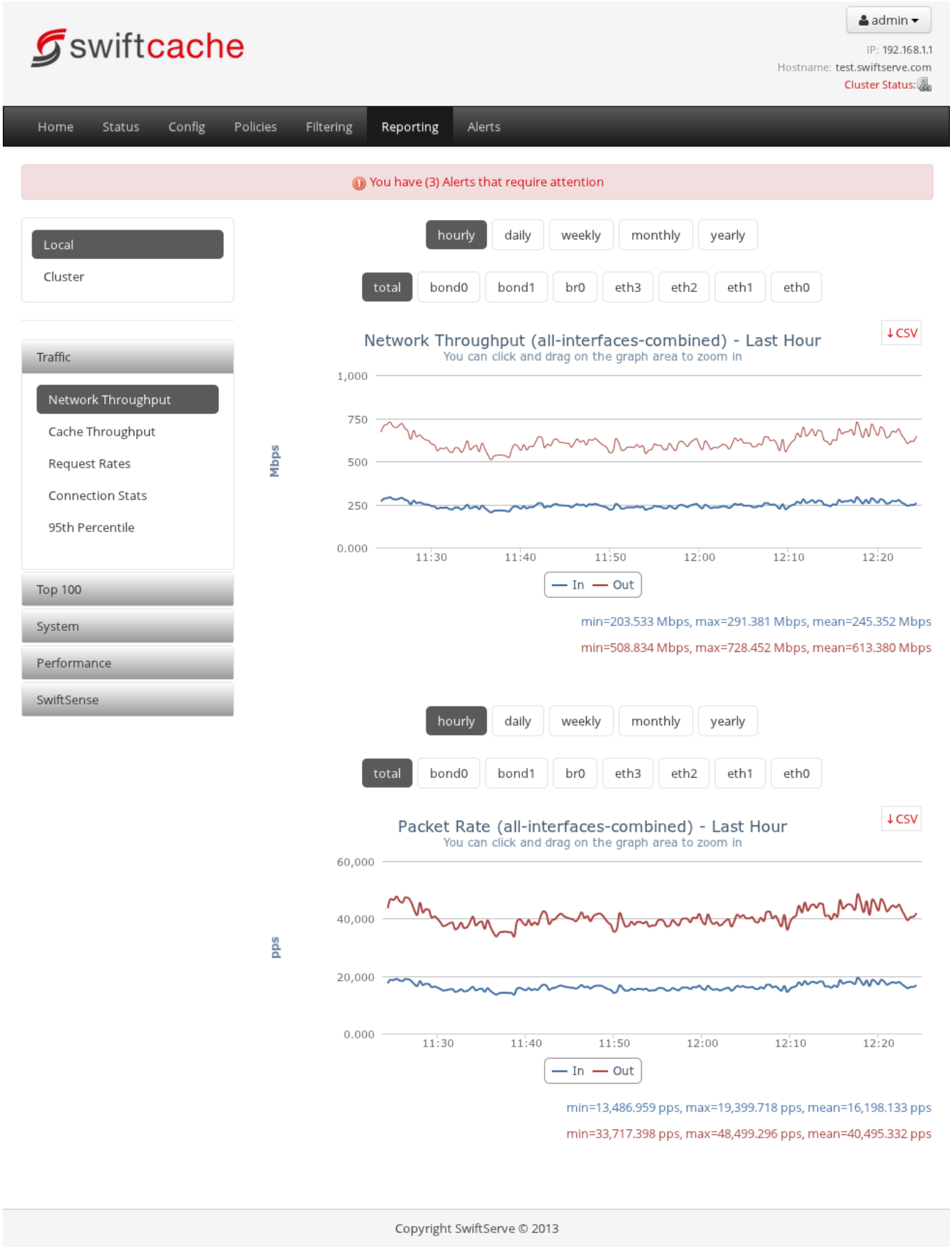
Many of the graphs displayed under the *Reporting* tab are interactive. Depending on the type of graph shown it is possible to:

- select hourly to yearly scales as a starting point;
- zoom into a particular time period;
- revert the view by clicking on *Reset zoom*;
- disable one or more of the graph parameters by clicking on them;
- re-enable disabled parameters by clicking on them (disabled parameters are shown greyed-out).

Traffic

The *Traffic* subsection shows network information displayed on four pages, which are described in more detail below.

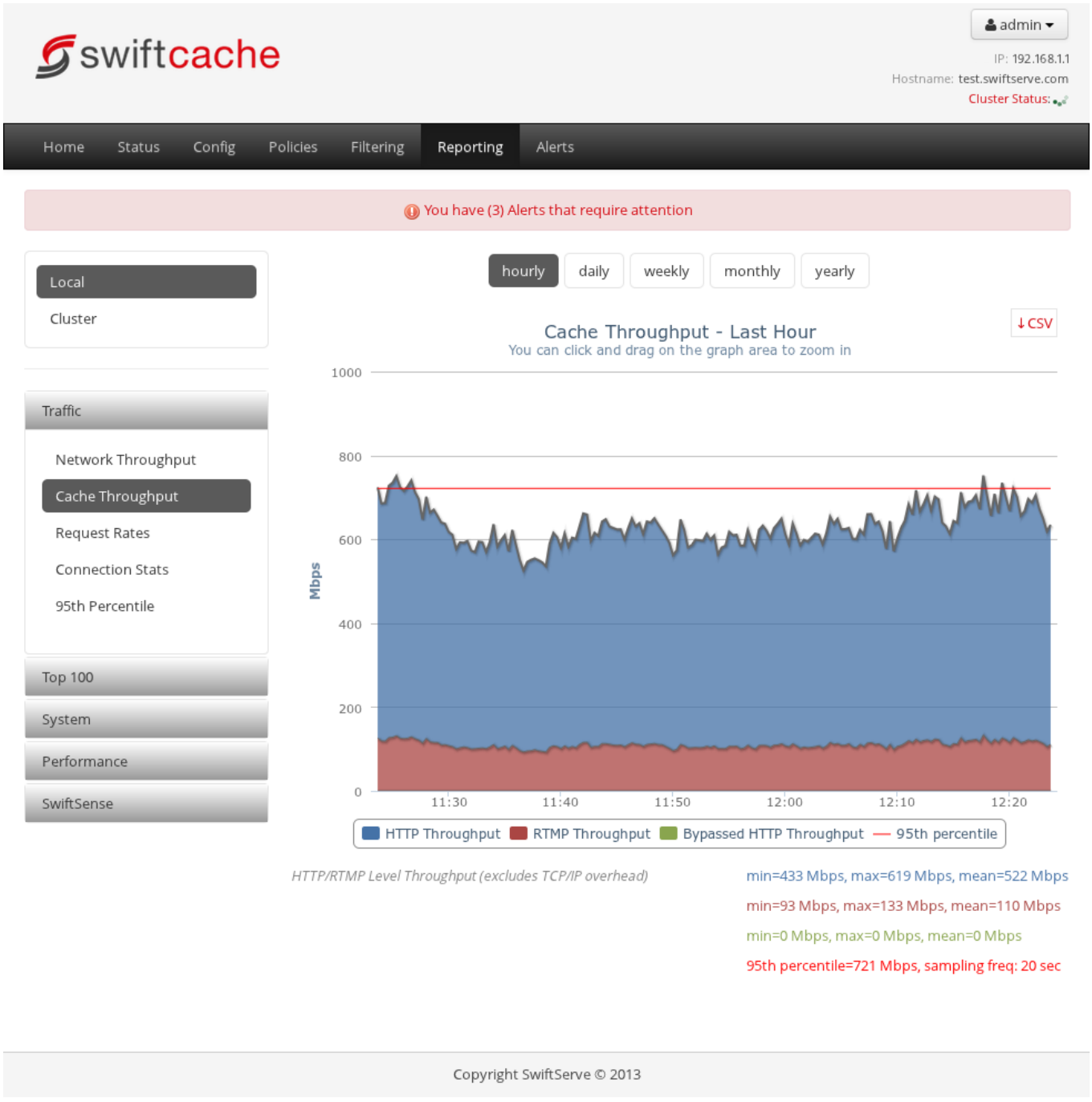
Network Throughput



The graphs show network throughput and packet rate for the total and for individual interfaces. These graphs show the actual data that is seen on the wire, including protocol overheads. The difference between the *in* and

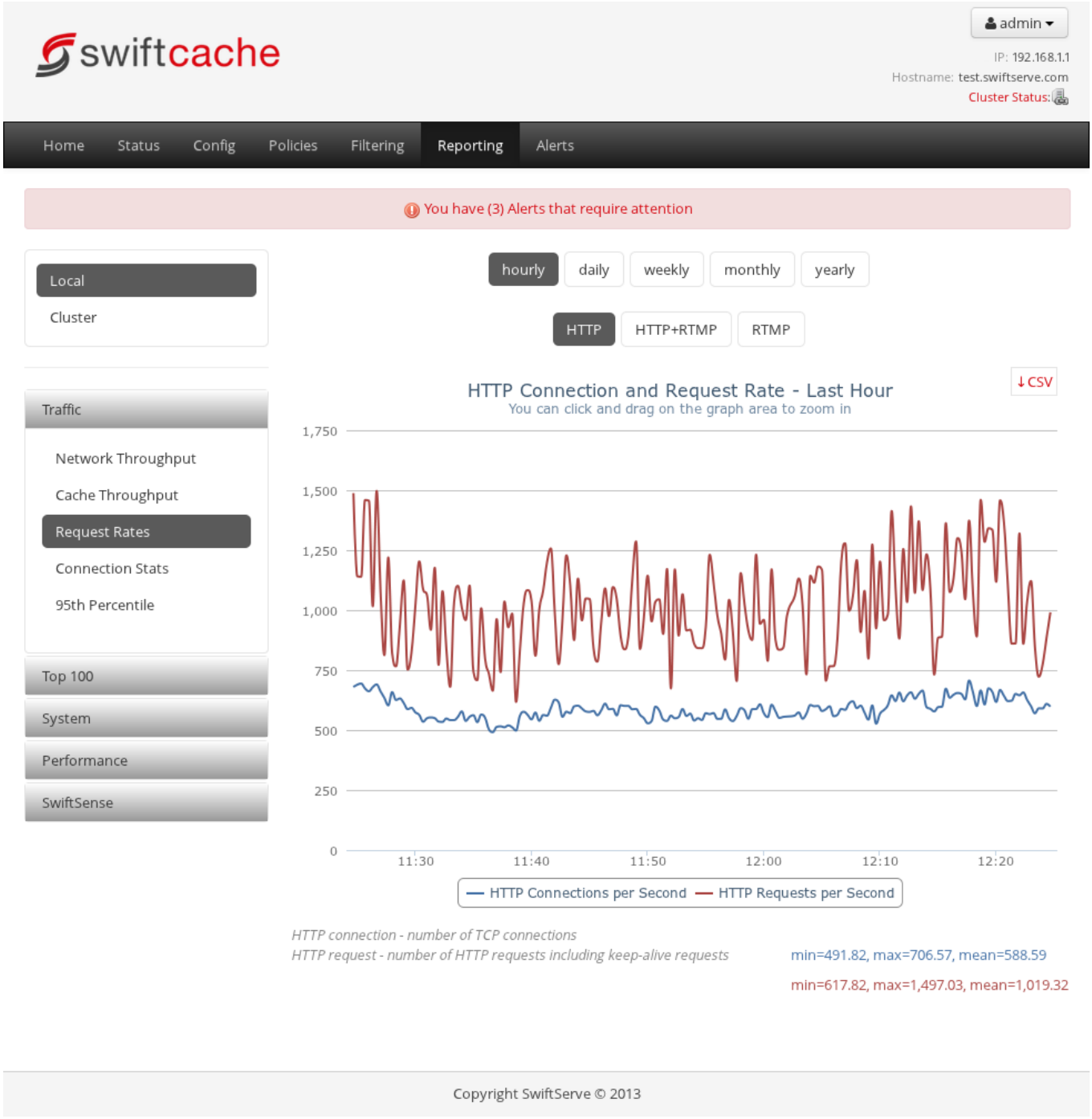
out lines corresponds to the data that is served from cache, namely the bandwidth saving.

Cache Throughput



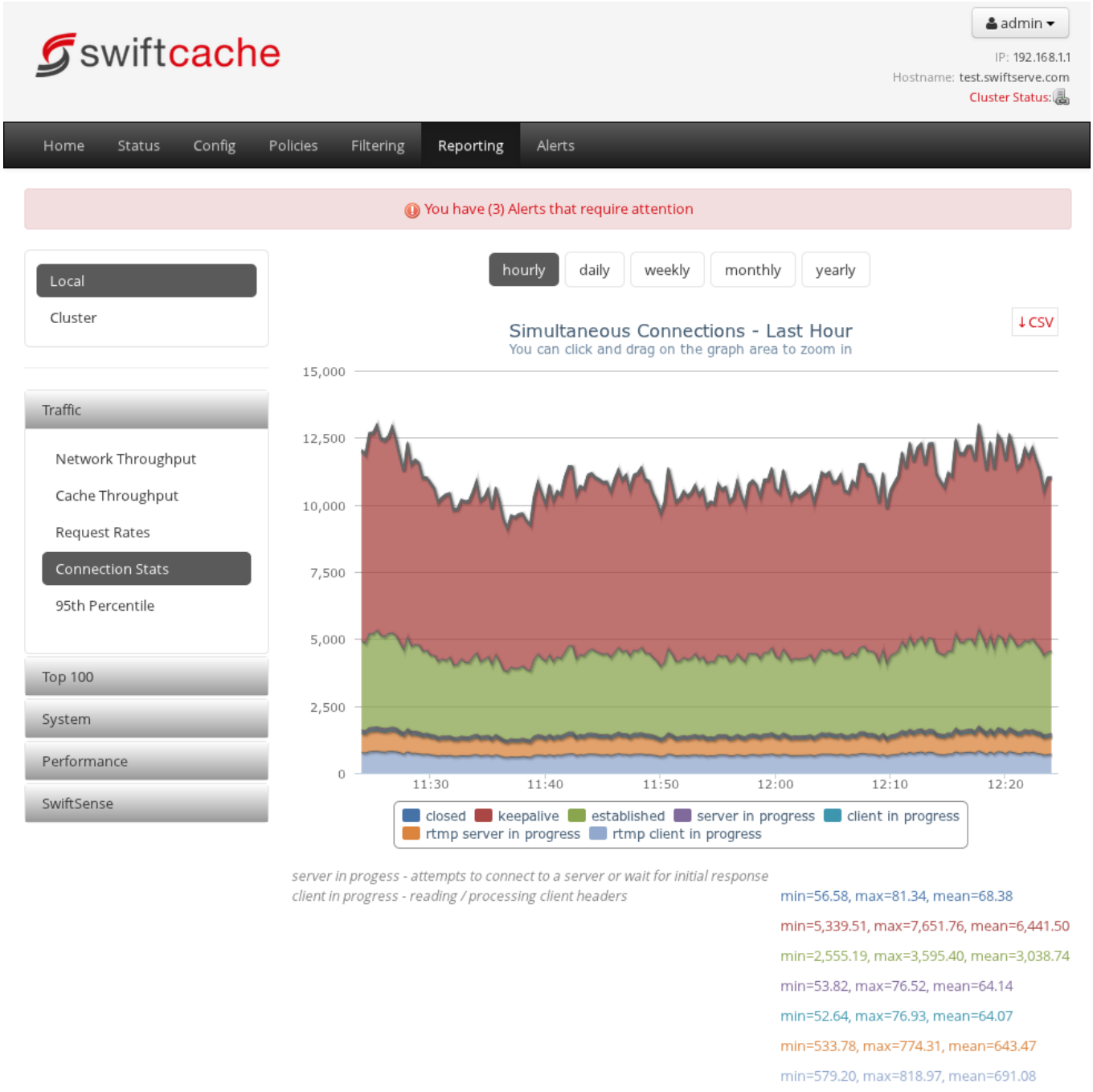
The graphs show HTTP throughput, excluding TCP/IP overhead. If RTMP is enabled, graphs of RTMP throughput are also shown. These graphs are particularly important for assessing the performance of SwiftCache.

Request Rates



The graphs show the number of HTTP connections and requests per second. If RTMP is enabled, graphs of RTMP throughput are also shown. The HTTP request rate is one of the most reliable indicators of load on SwiftCache as it equates almost directly to Disk Input/Output (IO), which is one of the resource limitations of the SwiftCache.

Connection Stats



The graphs show the number of simultaneous connections, grouped by their connection state: *Keep-Alive*, *Established*, *Client In Progress*, *Server In Progress* and *Closed*. The number of simultaneous connections is one of the parameters utilised by the overload protection mechanism.

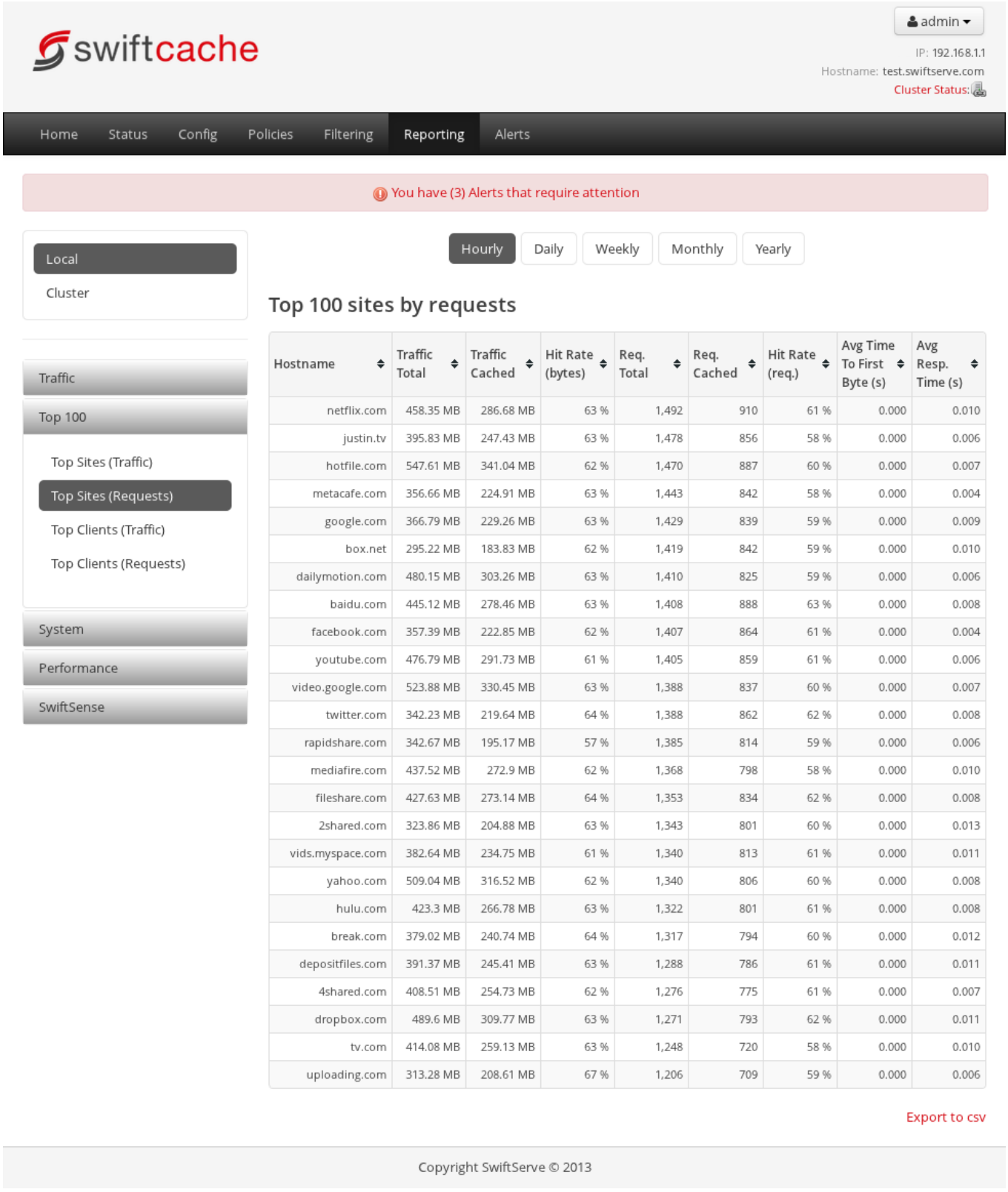
Top 100

This subsection shows tables of information on the top 100 sites and clients by amount of traffic and number of requests, updated every five minutes. This allows an operator to determine the most heavy users and frequently-accessed sites. This information can then guide the operator when tuning the SwiftCache configuration and policies for optimal performance.

Each table provides the following information: *Traffic Total*, *Traffic Cached*, *Hit Rate (traffic)*, *Requests Total*,

Requests Cached, *Hit Rate (requests)*, *Average Time To First Byte* and *Average Response Time*. The information can also be exported as a Comma-Separated Values (CSV) file.

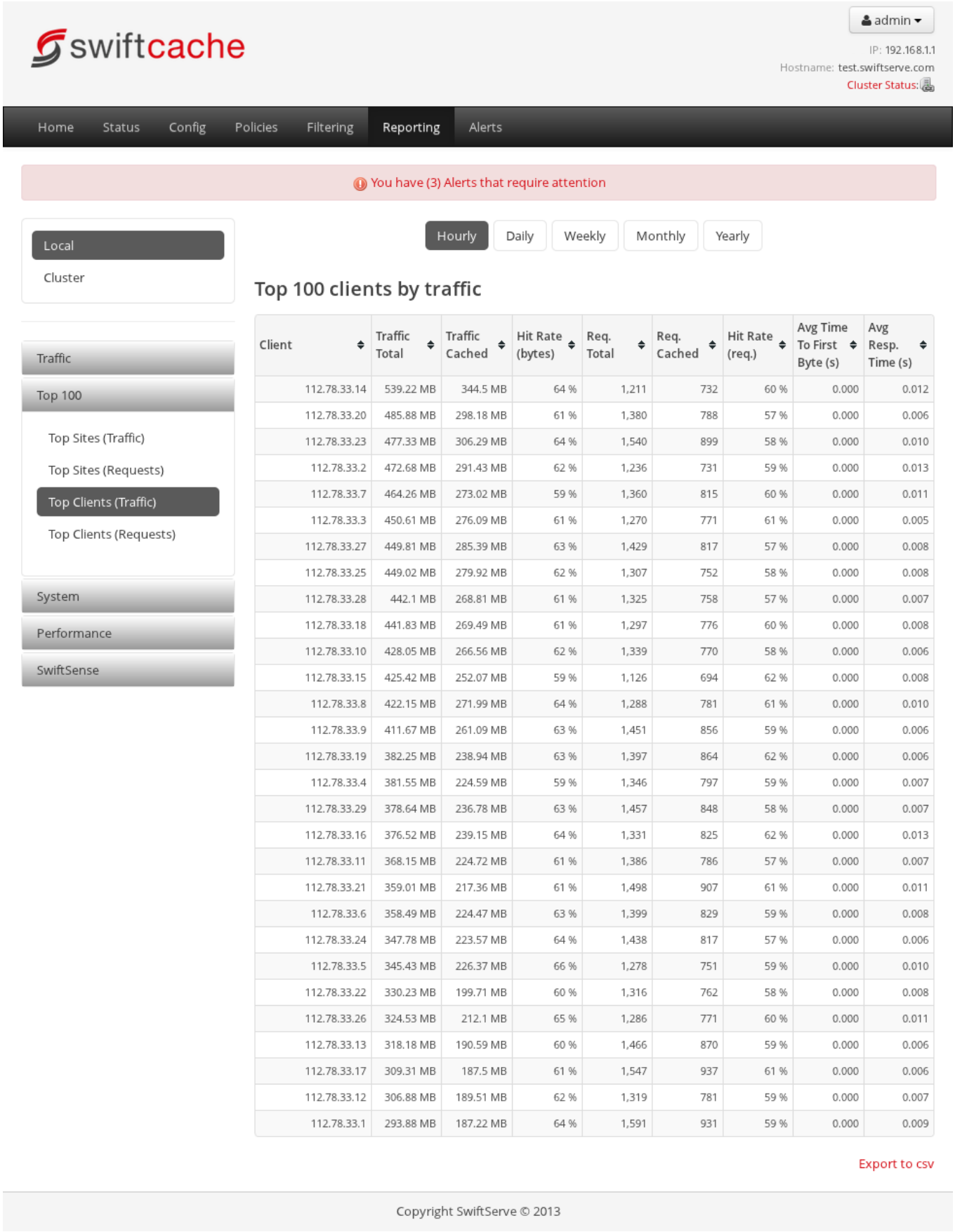
Top Sites (Requests)



Hostname	Traffic Total	Traffic Cached	Hit Rate (bytes)	Req. Total	Req. Cached	Hit Rate (req.)	Avg Time To First Byte (s)	Avg Resp. Time (s)
netflix.com	458.35 MB	286.68 MB	63 %	1,492	910	61 %	0.000	0.010
justin.tv	395.83 MB	247.43 MB	63 %	1,478	856	58 %	0.000	0.006
hotfile.com	547.61 MB	341.04 MB	62 %	1,470	887	60 %	0.000	0.007
metacafe.com	356.66 MB	224.91 MB	63 %	1,443	842	58 %	0.000	0.004
google.com	366.79 MB	229.26 MB	63 %	1,429	839	59 %	0.000	0.009
box.net	295.22 MB	183.83 MB	62 %	1,419	842	59 %	0.000	0.010
dailymotion.com	480.15 MB	303.26 MB	63 %	1,410	825	59 %	0.000	0.006
baidu.com	445.12 MB	278.46 MB	63 %	1,408	888	63 %	0.000	0.008
facebook.com	357.39 MB	222.85 MB	62 %	1,407	864	61 %	0.000	0.004
youtube.com	476.79 MB	291.73 MB	61 %	1,405	859	61 %	0.000	0.006
video.google.com	523.88 MB	330.45 MB	63 %	1,388	837	60 %	0.000	0.007
twitter.com	342.23 MB	219.64 MB	64 %	1,388	862	62 %	0.000	0.008
rapidshare.com	342.67 MB	195.17 MB	57 %	1,385	814	59 %	0.000	0.006
mediafire.com	437.52 MB	272.9 MB	62 %	1,368	798	58 %	0.000	0.010
fileshare.com	427.63 MB	273.14 MB	64 %	1,353	834	62 %	0.000	0.008
2shared.com	323.86 MB	204.88 MB	63 %	1,343	801	60 %	0.000	0.013
vids.myspace.com	382.64 MB	234.75 MB	61 %	1,340	813	61 %	0.000	0.011
yahoo.com	509.04 MB	316.52 MB	62 %	1,340	806	60 %	0.000	0.008
hulu.com	423.3 MB	266.78 MB	63 %	1,322	801	61 %	0.000	0.008
break.com	379.02 MB	240.74 MB	64 %	1,317	794	60 %	0.000	0.012
depositfiles.com	391.37 MB	245.41 MB	63 %	1,288	786	61 %	0.000	0.011
4shared.com	408.51 MB	254.73 MB	62 %	1,276	775	61 %	0.000	0.007
dropbox.com	489.6 MB	309.77 MB	63 %	1,271	793	62 %	0.000	0.011
tv.com	414.08 MB	259.13 MB	63 %	1,248	720	58 %	0.000	0.010
uploading.com	313.28 MB	208.61 MB	67 %	1,206	709	59 %	0.000	0.006

The table shows the top 100 sites according to the number of requests for each. This shows most popular sites accessed by clients.

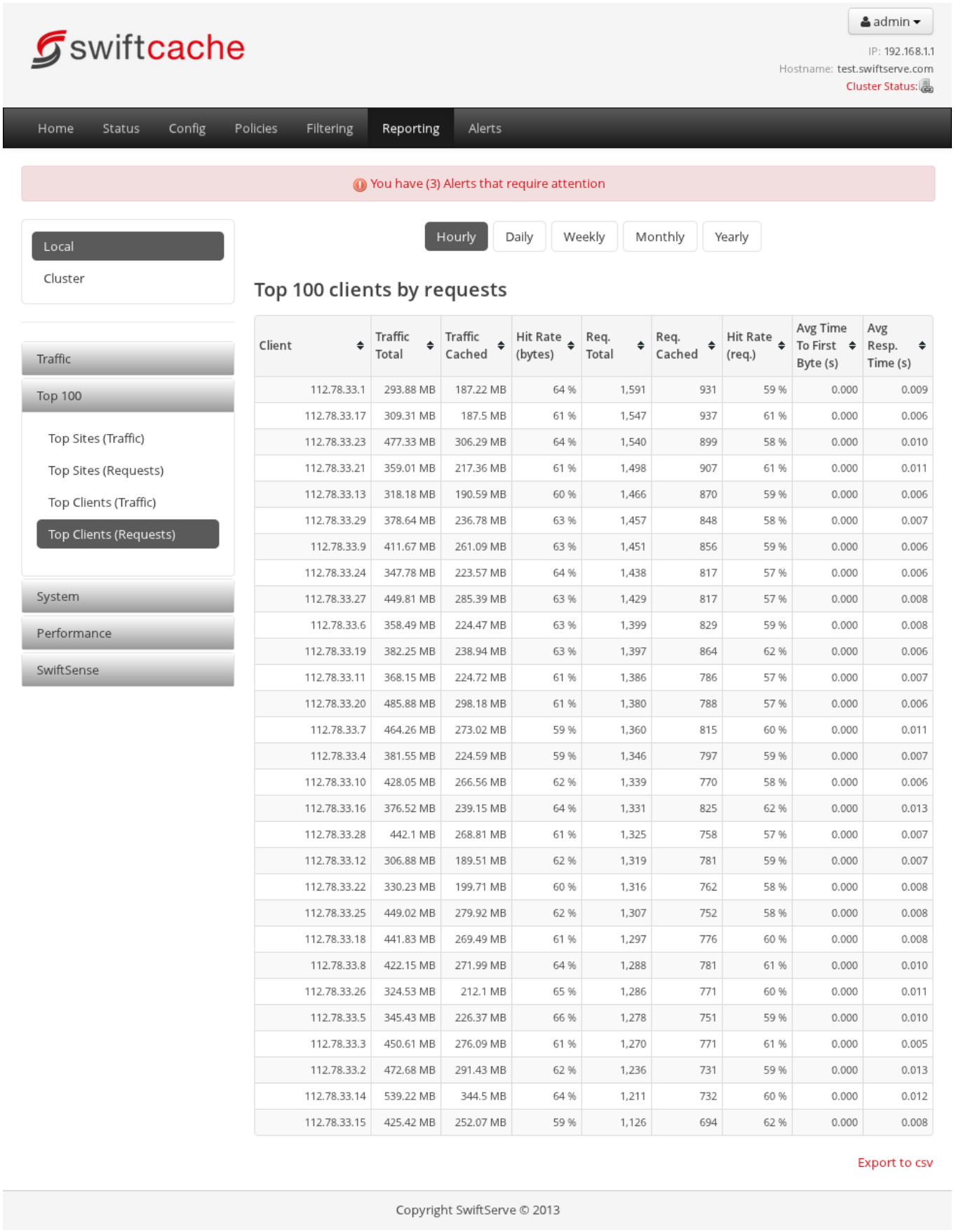
Top Clients (Traffic)



The table shows the top 100 clients according to the amount of network traffic for each. This shows which clients are using the most bandwidth. Note that it is possible that many end-users may in fact be behind a single client IP address, for example in a corporate environment where all client requests pass via a forwarding web proxy

that is not operating fully-transparently.

Top Clients (Requests)



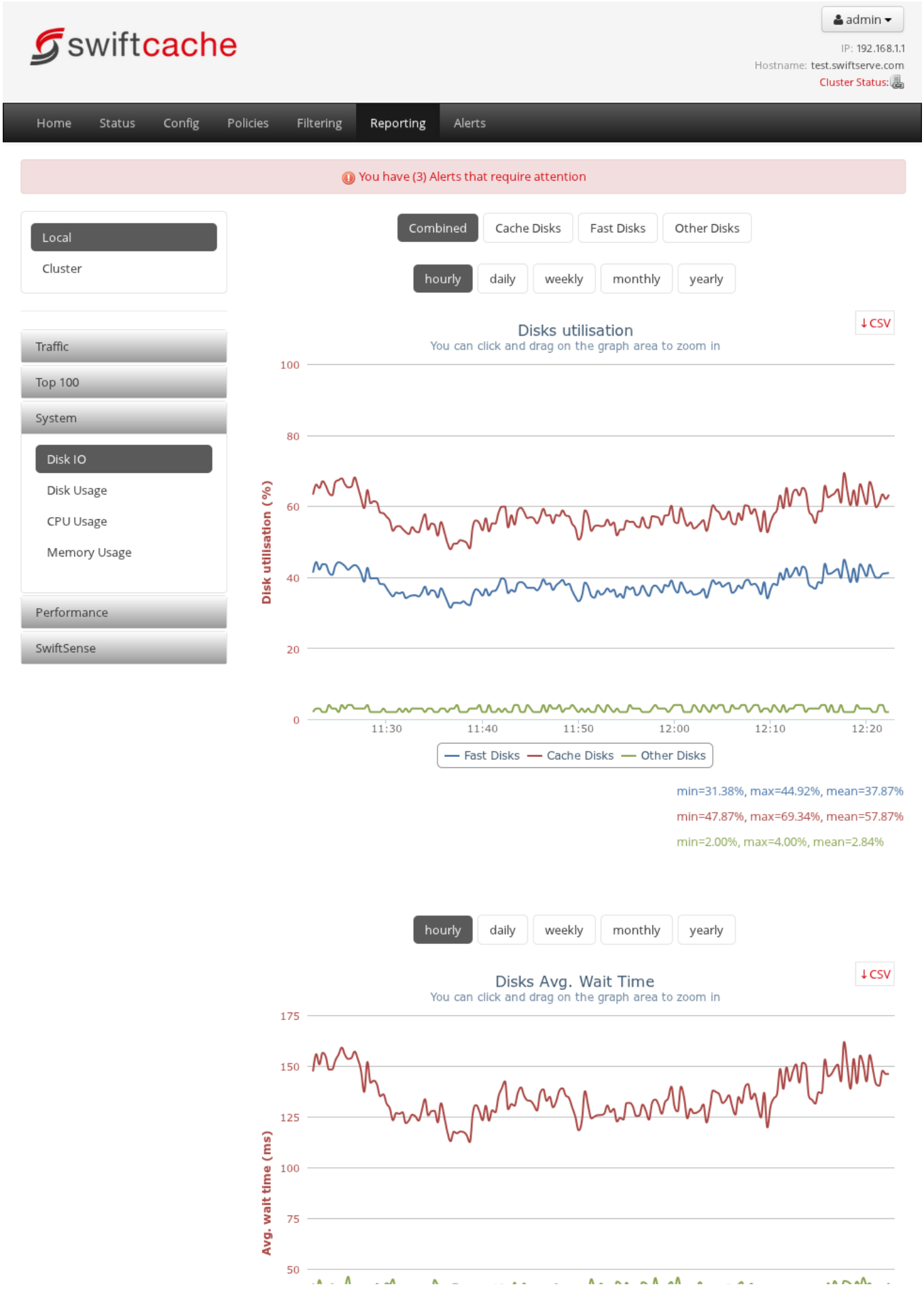
Client	Traffic Total	Traffic Cached	Hit Rate (bytes)	Req. Total	Req. Cached	Hit Rate (req.)	Avg Time To First Byte (s)	Avg Resp. Time (s)
112.78.33.1	293.88 MB	187.22 MB	64 %	1,591	931	59 %	0.000	0.009
112.78.33.17	309.31 MB	187.5 MB	61 %	1,547	937	61 %	0.000	0.006
112.78.33.23	477.33 MB	306.29 MB	64 %	1,540	899	58 %	0.000	0.010
112.78.33.21	359.01 MB	217.36 MB	61 %	1,498	907	61 %	0.000	0.011
112.78.33.13	318.18 MB	190.59 MB	60 %	1,466	870	59 %	0.000	0.006
112.78.33.29	378.64 MB	236.78 MB	63 %	1,457	848	58 %	0.000	0.007
112.78.33.9	411.67 MB	261.09 MB	63 %	1,451	856	59 %	0.000	0.006
112.78.33.24	347.78 MB	223.57 MB	64 %	1,438	817	57 %	0.000	0.006
112.78.33.27	449.81 MB	285.39 MB	63 %	1,429	817	57 %	0.000	0.008
112.78.33.6	358.49 MB	224.47 MB	63 %	1,399	829	59 %	0.000	0.008
112.78.33.19	382.25 MB	238.94 MB	63 %	1,397	864	62 %	0.000	0.006
112.78.33.11	368.15 MB	224.72 MB	61 %	1,386	786	57 %	0.000	0.007
112.78.33.20	485.88 MB	298.18 MB	61 %	1,380	788	57 %	0.000	0.006
112.78.33.7	464.26 MB	273.02 MB	59 %	1,360	815	60 %	0.000	0.011
112.78.33.4	381.55 MB	224.59 MB	59 %	1,346	797	59 %	0.000	0.007
112.78.33.10	428.05 MB	266.56 MB	62 %	1,339	770	58 %	0.000	0.006
112.78.33.16	376.52 MB	239.15 MB	64 %	1,331	825	62 %	0.000	0.013
112.78.33.28	442.1 MB	268.81 MB	61 %	1,325	758	57 %	0.000	0.007
112.78.33.12	306.88 MB	189.51 MB	62 %	1,319	781	59 %	0.000	0.007
112.78.33.22	330.23 MB	199.71 MB	60 %	1,316	762	58 %	0.000	0.008
112.78.33.25	449.02 MB	279.92 MB	62 %	1,307	752	58 %	0.000	0.008
112.78.33.18	441.83 MB	269.49 MB	61 %	1,297	776	60 %	0.000	0.008
112.78.33.8	422.15 MB	271.99 MB	64 %	1,288	781	61 %	0.000	0.010
112.78.33.26	324.53 MB	212.1 MB	65 %	1,286	771	60 %	0.000	0.011
112.78.33.5	345.43 MB	226.37 MB	66 %	1,278	751	59 %	0.000	0.010
112.78.33.3	450.61 MB	276.09 MB	61 %	1,270	771	61 %	0.000	0.005
112.78.33.2	472.68 MB	291.43 MB	62 %	1,236	731	59 %	0.000	0.013
112.78.33.14	539.22 MB	344.5 MB	64 %	1,211	732	60 %	0.000	0.012
112.78.33.15	425.42 MB	252.07 MB	59 %	1,126	694	62 %	0.000	0.008

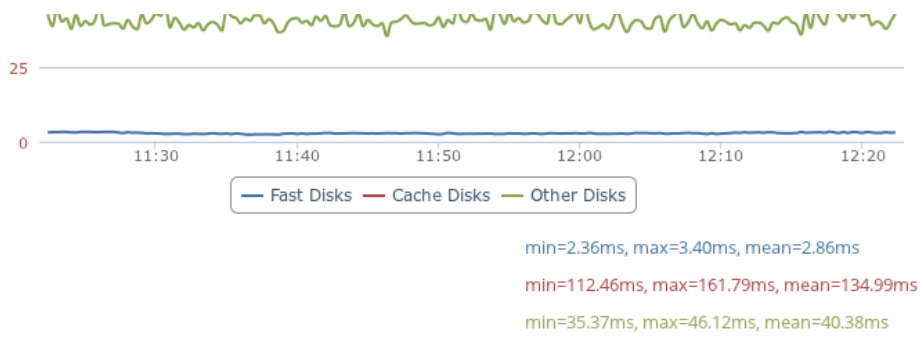
A table is shown of the top 100 clients according to the number of requests. This shows which clients make the most requests. Again, many end-users may in fact be behind a single client IP address, as noted above.

System

The *System* subsection shows information on the level of utilisation of hardware resources by SwiftCache. It is divided into pages corresponding to *Disk IO* (input/output), *Disk Usage*, *CPU* and *Memory*.

Disk IO


swiftcache
admin
IP: 192.168.1.1
Hostname: test.swiftserve.com
Cluster Status:
Home
Status
Config
Policies
Filtering
Reporting
Alerts
You have (3) Alerts that require attention
Local
Cluster
Traffic
Top 100
System
Disk IO
Disk Usage
CPU Usage
Memory Usage
Performance
SwiftSense
Combined
Cache Disks
Fast Disks
Other Disks
hourly
daily
weekly
monthly
yearly
↓CSV
Disks utilisation
You can click and drag on the graph area to zoom in
Disk utilisation (%)
100
80
60
40
20
0
11:30
11:40
11:50
12:00
12:10
12:20
Fast Disks
Cache Disks
Other Disks
min=31.38%, max=44.92%, mean=37.87%
min=47.87%, max=69.34%, mean=57.87%
min=2.00%, max=4.00%, mean=2.84%
hourly
daily
weekly
monthly
yearly
↓CSV
Disks Avg. Wait Time
You can click and drag on the graph area to zoom in
Avg. wait time (ms)
175
150
125
100
75
50



Copyright SwiftServe © 2013

The *Disk IO* page displays the number of read and write operations that the SwiftCache appliance is currently handling. On most systems it is expected to see a higher number of disk writes than disk reads, especially on fresh installs where the disks are not yet full.

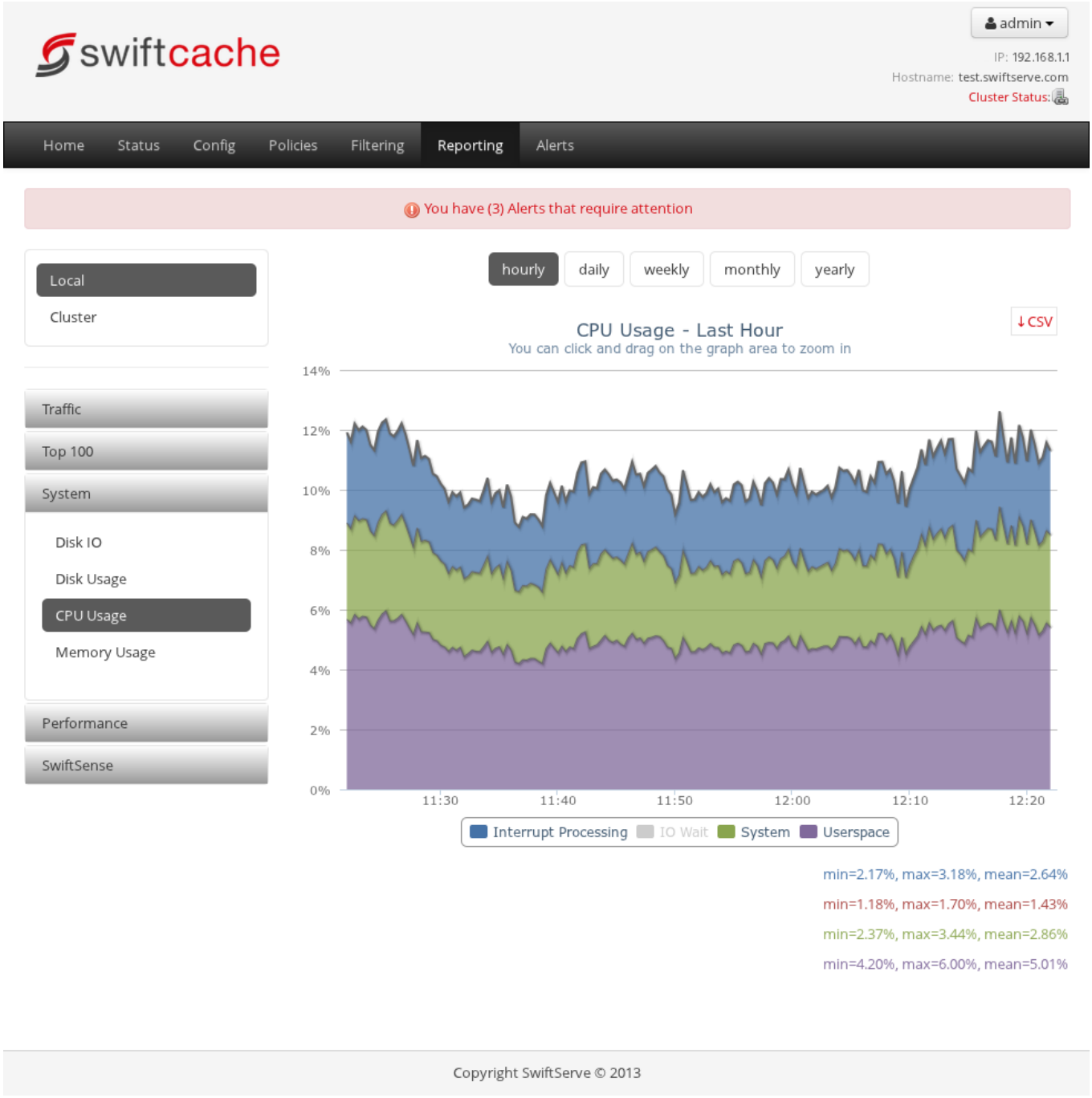
Further information is provided under sub-tabs as interactive graphs.

Combined	Shows graphs of the disk utilisation percentage and wait times for each of the disk types.
Cache Disks	Shows graphs of the read and write transactions per second for each of the cache disks, and graphs of the disk utilisation percentage and wait times for each of the cache disks.
Fast Disks	Shows graphs of the read and write transactions per second for each of the fast disks (Solid State Disks or *SSDs*), and graphs of the disk utilisation percentage and wait times for each of the fast disks (SSDs).
Other Disks	Shows graphs of the read and write transactions per second for each of the other disks, and graphs of the disk utilisation percentage and wait times for each of the other disks.

Disk Usage

This bar chart indicates the percentage usage of each of the cache and fast disks. Under normal appliance operating conditions the disk usage will intentionally be high; the cache cleaning algorithms will maintain the disk usage at the `cache_max_usage` threshold (set by default at 90 percent) and new content will replace old.

CPU Usage



This interactive graph shows the CPU usage across all cores of the SwiftCache CPUs for each of the *Interrupt Processing*, *IO Wait*, *System* and *Userspace* processes.
Note that the *IO Wait* category is idle CPU time where there were jobs waiting for disk IO. It does not indicate CPU usage and should be considered a subset of the free CPU capacity. For this reason the parameter is disabled by default, but can be enabled by clicking on it.

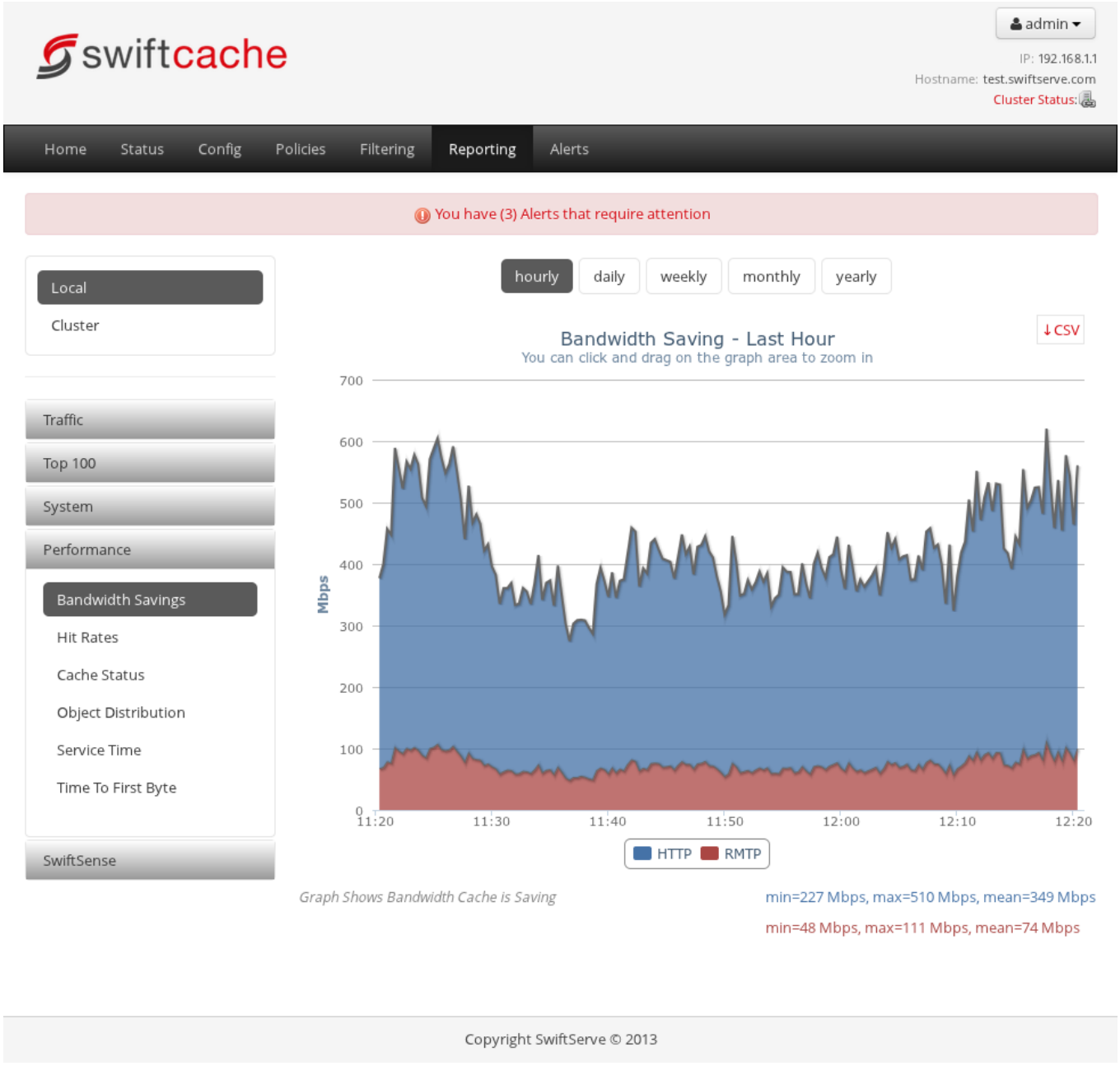
Memory Usage

This interactive graph shows the memory *Used*, *Free* and *Buffer Cache*. Any memory that is not used by the SwiftCache processes is typically allocated to the disk buffer cache.

Performance

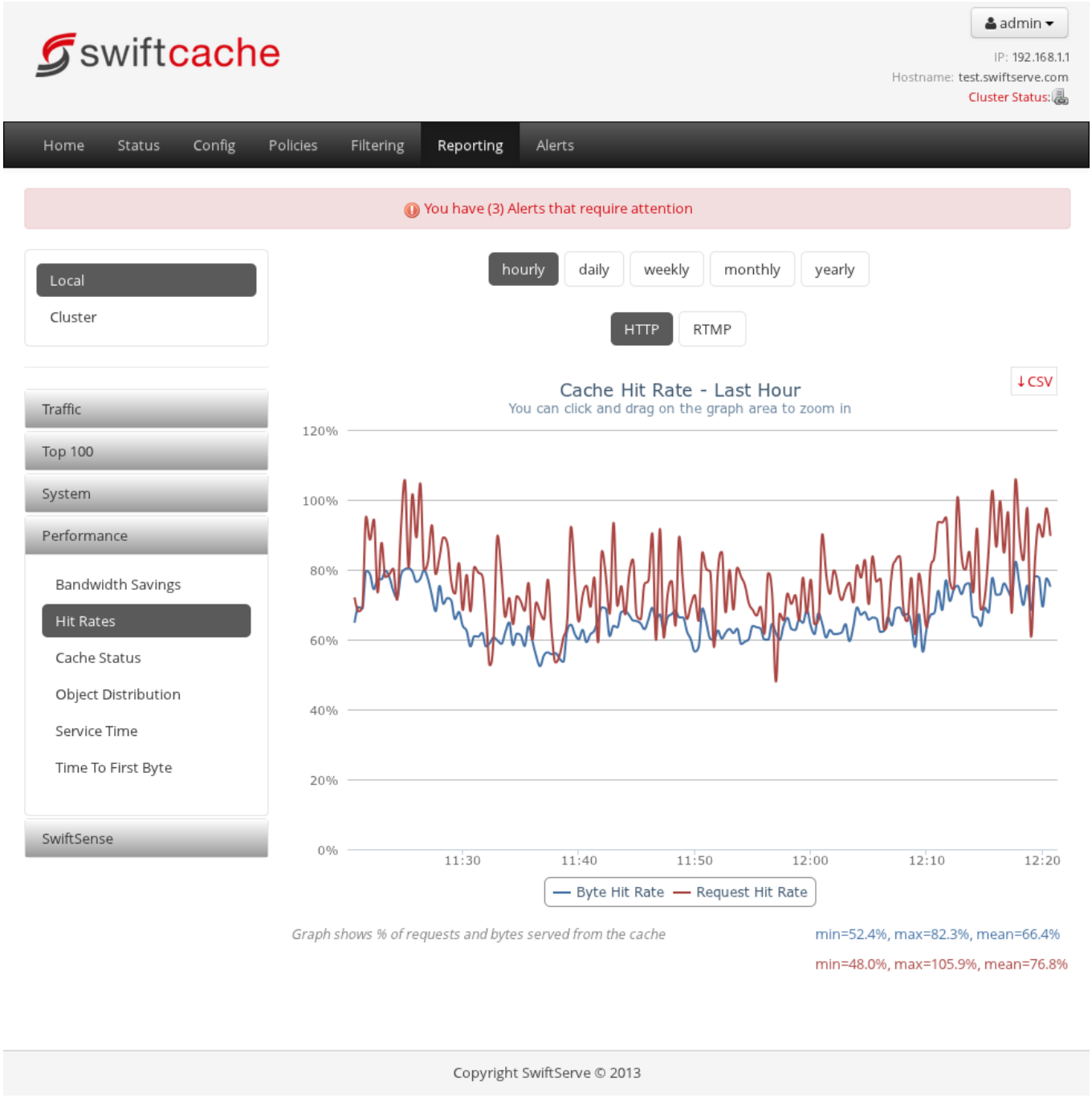
This subsection shows information on the performance of the SwiftCache appliance; it is divided into pages corresponding to *Bandwidth Savings*, *Hit Rates*, *Cache Status*, *Object Distribution*, *Service Time*, *Time To First Byte* and *Cache Acceleration*.

Bandwidth Savings



This graph shows the amount of the HTTP bandwidth saved. Bandwidth saved is calculated as the difference between the amount of network traffic sent to clients and traffic received from the origin servers, less the protocol overhead (approximately 10 percent).

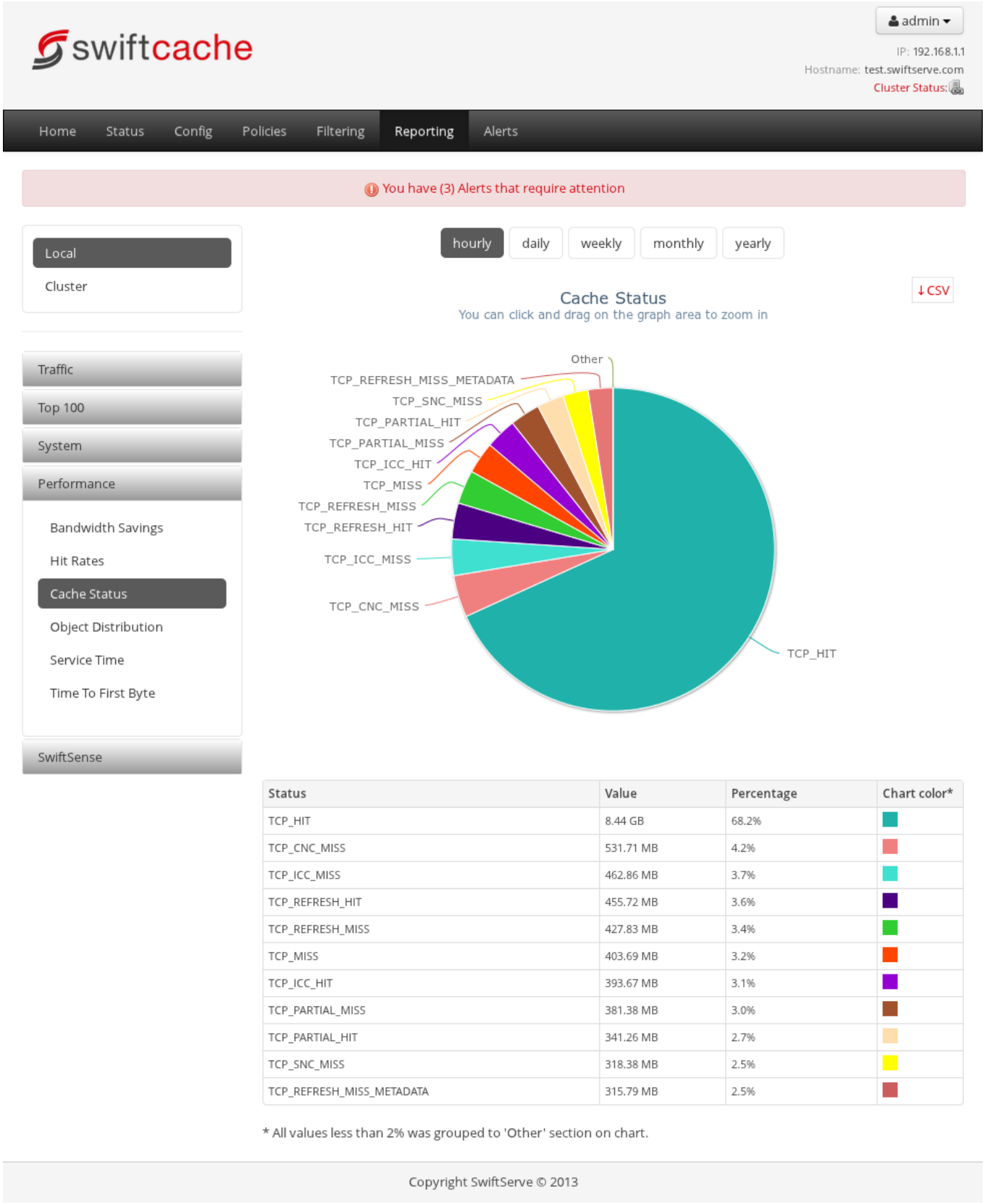
Hit Rates



This interactive graph shows the percentage byte and request hit rates achieved by the SwiftCache appliance. The hit rates are particularly useful indicators of the efficiency of SwiftCache.

Byte Hit Rate	Records the overall percentage of application traffic that has been served from the cache (where application traffic excludes the protocol overheads).
Request Hit Rate	Shows the percentage of requests that are served from cache.

Cache Status



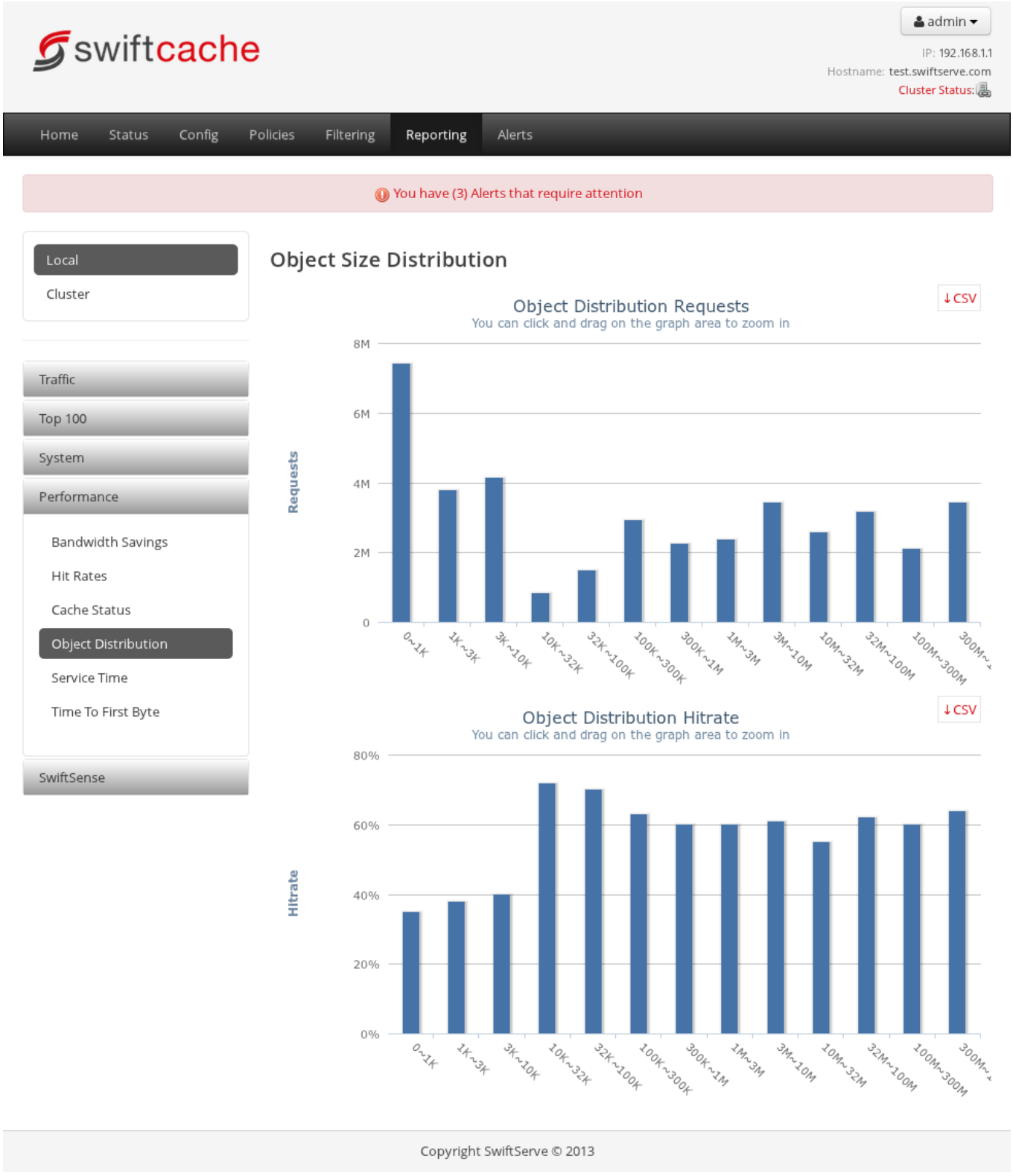
This pie chart shows the relative proportions of different cache response codes for the client requests. This highlights the relative performance of the cache, since the objective is to have as large a portion as possible of *TCP_HIT* and *TCP_PARTIAL/REFRESH_HIT* responses.

Underneath there is a table of the corresponding raw data, showing the amount of data transferred for each response code and the percentage of the total bytes.

The most common cache response codes are described below:

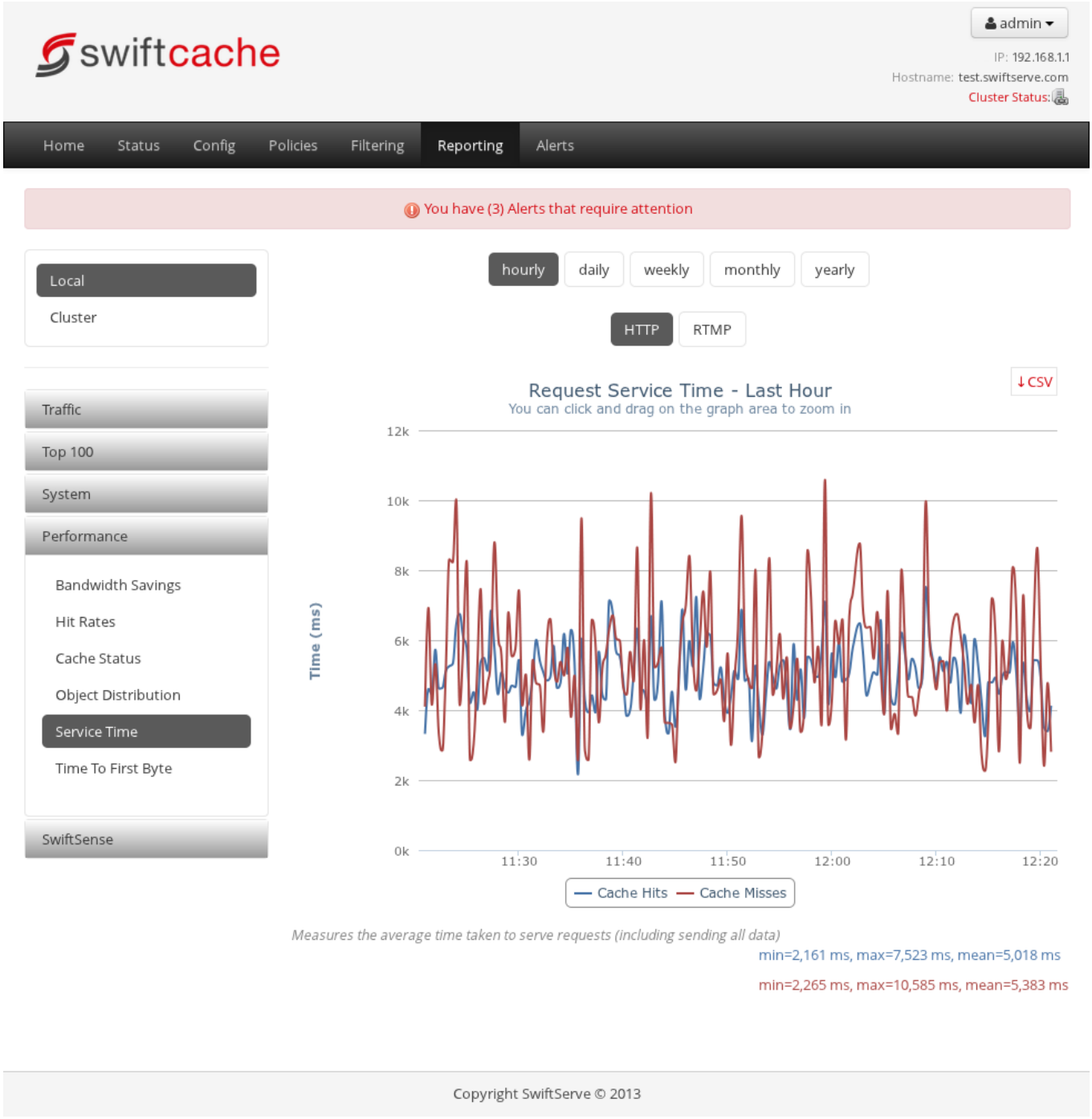
TCP_HIT	The object was served from the cache.
TCP_PARTIAL_HIT	The object was partially served from the cache.
TCP_REFRESH_HIT	The object was served from the cache after confirming it was still valid.
TCP_MISS	The object was not previously seen and so it was retrieved from the origin server.
TCP_CNC_MISS	The client specified not to use the cache (and so the object was retrieved from the origin server).
TCP_SNC_MISS	The origin server had specified not to use the cache (and so the object was retrieved from it).
TCP_REFRESH_MISS	The object was retrieved from the origin server since it was confirmed to have changed since the last time it was stored.

Object Distribution



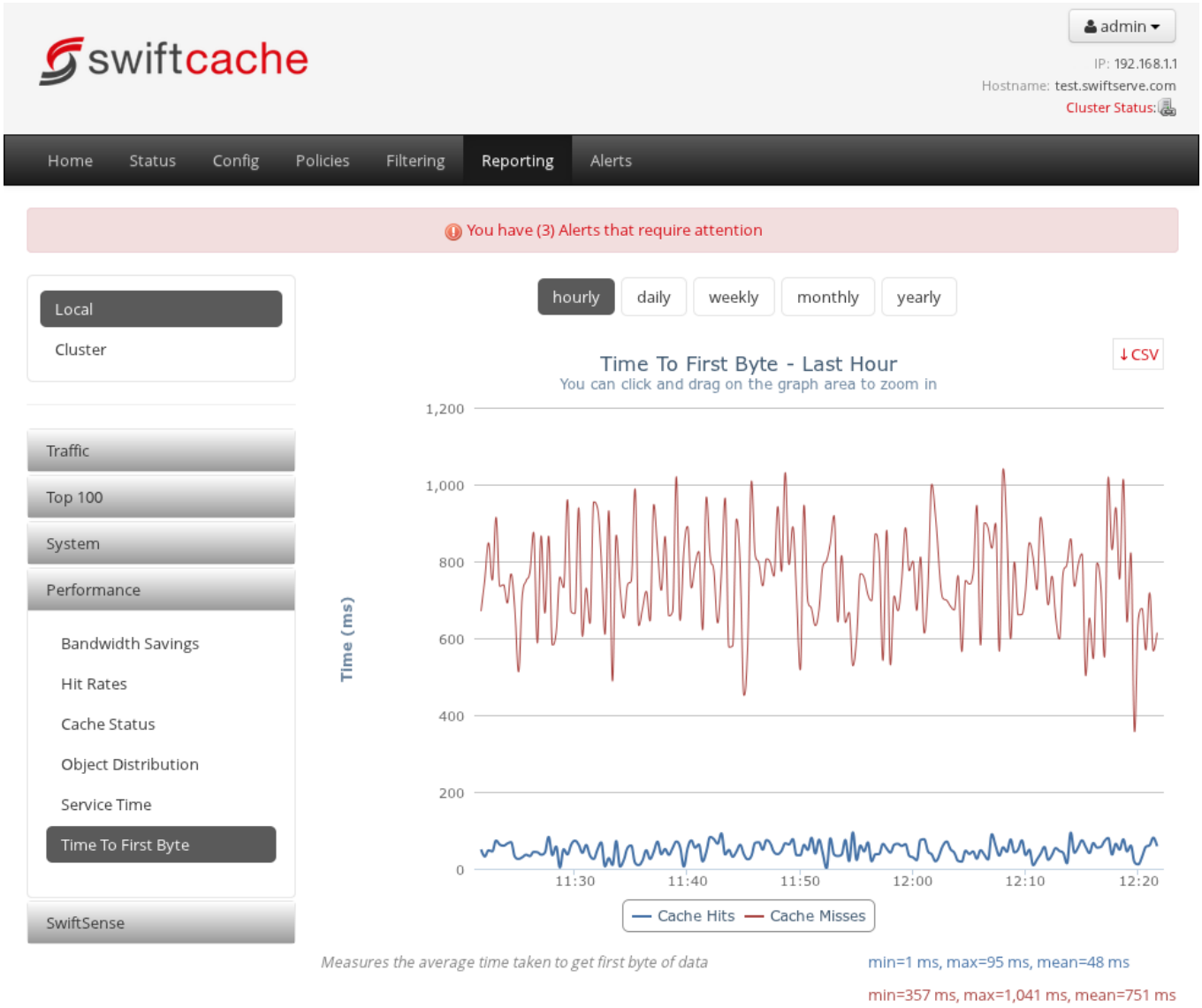
These charts show the requests and hitrate percentages according to the profile of different-sized objects in the cache. This graph is very useful for analysing the traffic distribution that the SwiftCache is serving.

Service Time



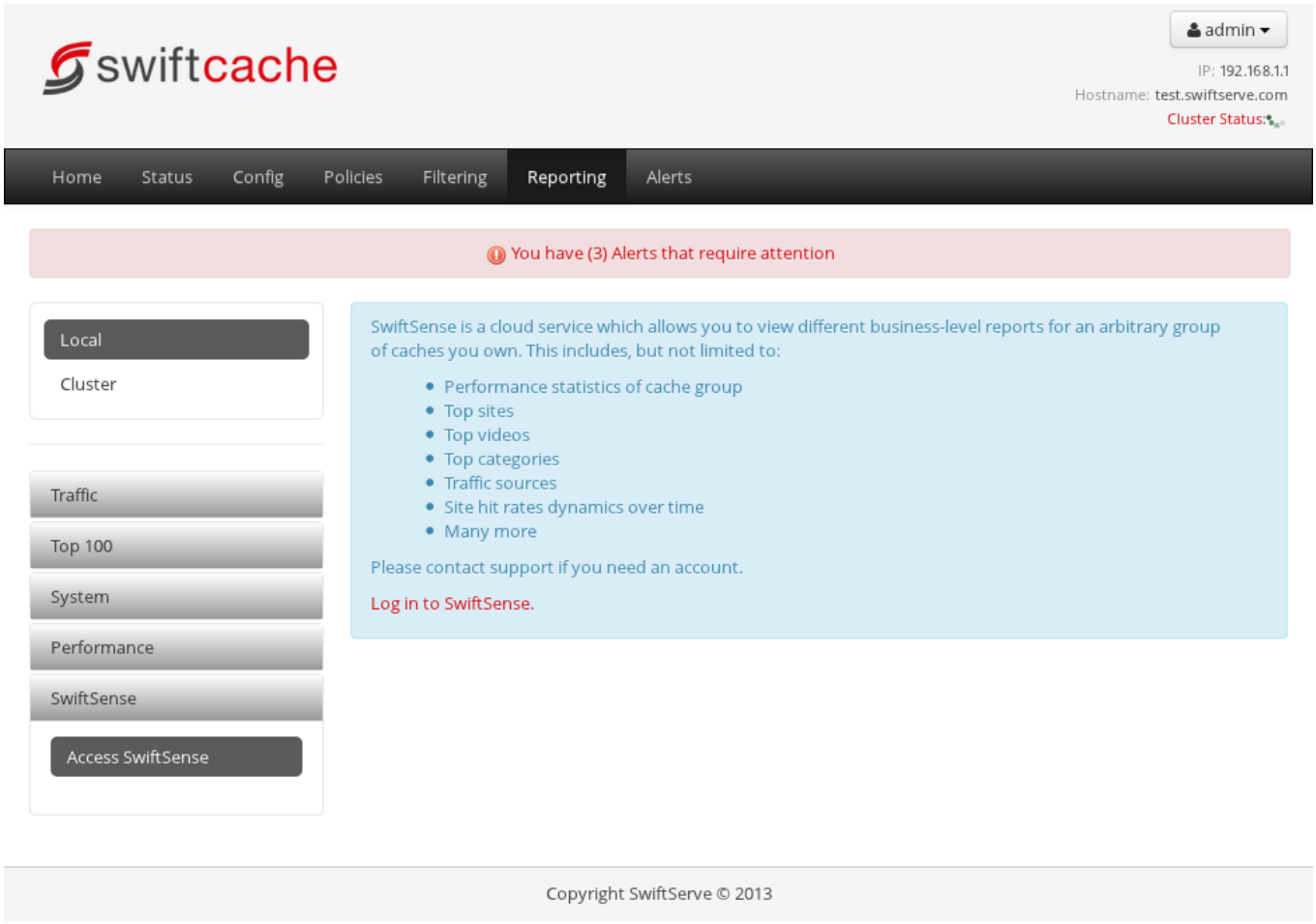
This graph shows the average service time for client requests (the time taken including sending all the data), for both cache hits and cache misses. The *Cache Miss* line shows the time to complete requests by connecting back to the origin server, while the *Cache Hit* line shows the time to complete requests that are served from cache.

Time To First Byte



This graph shows the average time for client requests to begin to be served, namely the time taken for clients to receive the first byte of data, for both cache hits and cache misses.

SwiftSense



SwiftSense is a separate component to SwiftCache. This subsection outlines the information available in SwiftSense and provides a link to login. Please see the **SwiftSense** section later in this manual for more detail.

8.4.7 Alerts

The *Alerts* tab allows operators to view alerts and manage notifications. For more information on alerts and notification, please refer to the **Alerts** section in the **Monitoring** chapter of this manual.

Viewing and Acknowledging Alerts

By default the *Alerts* tab shows unacknowledged alerts for the appliance and cluster, if applicable, from the last seven days. For each alert, the number of times it has occurred and the severity are displayed in a table. To display a tooltip describing the alert in more detail and suggesting remedial actions, hover over the blue circle next to each alert.

To expand the table and show the individual occurrences, click on the name of the alert or the arrow in the *Actions* column. To acknowledge an individual occurrence of an alert, click on the green tick. Clicking on the green tick in the header will acknowledge all the alerts of that type.

Acknowledged alerts are hidden from the default view. The operator can acknowledge all current alerts by clicking on *Acknowledge All* in the bottom right-hand corner.

It is possible to display acknowledged alerts and the entire alert archive by using the checkboxes on the left-hand side. Free-text filtering can be achieved by typing into the text box above these options. To remove all current alert filters, click on the red circle.

Managing Notifications

Alert notifications can optionally be configured to be delivered via SNMP, HTTP GET, SysLog and Email (SMTP).

The default view of this is a table showing those notifications already configured.

Existing notifications can be enabled or disabled by toggling the checkbox in the *Enabled* column. To delete an individual notification, click on the red circle at the far right.

To add a new notification, select the action via the dropdown, which severities are to be included and the details according to the parameters displayed, then click the *Add Row* button. One or more new notifications can be added at a time. Finally, confirm and apply the new notification settings by clicking the *Save Changes* button.

Note that unsaved changes will be lost if the operator navigates away from the page before clicking the *Save Changes* button.

8.5 Cluster Settings

Almost every configuration item within SwiftCache has the ability to be shared within a cluster of SwiftCache nodes. Exceptions to this are configuration options specific to the appliance such as the appliance hostname and IP address.

8.5.1 Shared and Local Settings

There may be times when an operator would want to change a setting on a single SwiftCache without the change applying to other appliances in the cluster. For example, an operator may wish to test the effect of a setting safely or raise the level of logging detail on a single SwiftCache to debug a problem.

A setting configured for the scope of a single SwiftCache appliance is known as a *local* setting. A setting that applies to the whole SwiftCache cluster is known as a *shared* setting. Local settings take precedence (override) shared settings.

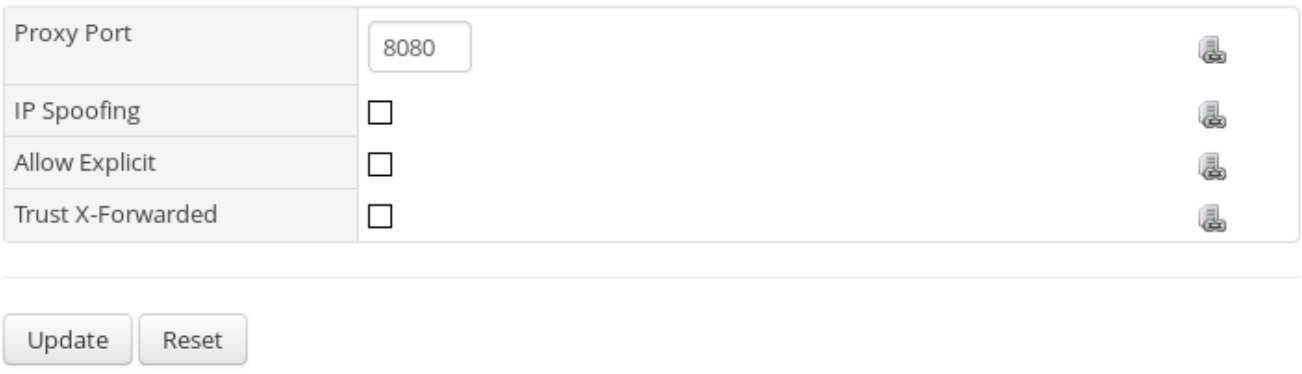
By default all settings are shared (noting the exceptions above). Even if a SwiftCache is operating by itself (i.e. there is only one appliance), by default it will have no local settings configured and so will use the shared settings by default.

When a SwiftCache is part of a cluster and a shared setting is changed on one node, then that setting will be automatically changed to the same value on every SwiftCache node within that cluster. If any local settings are configured on individual appliances, then these will continue to take precedence over the new shared setting.

To make it easy to tell if a configuration key is shared in a cluster or is local to the appliance, each configuration item that can be shared has an icon next to it showing whether it is shared or local.

A shared configuration key is denoted by an icon showing a server with a small chain link. A local configuration key is denoted by a server icon with no chain link. By default, an appliance operating by itself (i.e. there is only one appliance) will show the shared setting icon.

8.5.2 Example



In the example above, the *Proxy Port* option is using a local configuration setting, while the other options *IP*

Spoofing, *Allow Explicit* and *Trust-X-Forwarded* are all shared.

Clicking on the shared / local icon brings up a dialog that allows the operator to edit the shared and local settings. It is necessary to delete a local setting in order to revert to using the shared setting.

For more information on how SwiftCache controls configuration when operating in a cluster, please refer to the **Clustering** section of this document.

9 SwiftCache CLI

9.1 Overview

SwiftCache has a command-line interface (CLI) that can be accessed via SSH, direct console login or serial line connection. The CLI provides access to some advanced configuration options that are available only in the CLI and not in the web-based GUI.

Only the `admin` user account may access the CLI.

9.2 Applying Changes

Configuration changes are picked up automatically by SwiftCache so there is no need to save or commit before closing the CLI session.

- When changes to configuration are made via the CLI then they are generally applied immediately.
- When editing sections such as network settings, policies or filters for example, changes are applied only once the whole section has been completed and the `exit` command has been issued.

9.3 CLI Usage

9.3.1 Running Commands

When working with the CLI, instructions to SwiftCache, or *commands*, are given (*run*) to display or achieve something. Most commands require some specific detail (*arguments*), and these are typed after the command.

The command and arguments are typed at the CLI *prompt* and then the `[Enter]` key is pressed to run the command. The CLI prompt will be the hostname of your SwiftCache, followed by a `>`, for example:

```
yourswiftcache>
```

Note that in the following examples it is not needed to type the `yourswiftcache>` prompt itself, just the command and arguments that follow it.

For example, typing:

```
yourswiftcache> show allow_explicit
```

and then pressing `[Enter]` will run the `show` command with the argument `allow_explicit`. The `show` command will display the value of the SwiftCache configuration key you specify as the argument. In this example, it would display the value of the configuration key called `allow_explicit`.

9.3.2 Tab Completion

The SwiftCache CLI has a tab completion function to save time when entering commands. After typing the first letters of a command or argument, pressing the `[Tab]` key will cause the CLI to try and complete the rest. This will succeed if there is only one possible valid command or argument starting with those letters.

For example, typing:

```
yourswiftcache> sho
```

followed by the `[Tab]` key will complete the command to read `show`. This can be particularly useful with longer commands.

Pressing the `[Tab]` key twice at any point will display a list of the possible commands or arguments that can be completed. This can be useful in the event that an operator is familiar with the device but is not certain of the exact command or configuration key name.

From the default CLI prompt, pressing `[Tab]` twice will display all valid commands.

In the example below, the introductory text displayed on logging into the CLI is also shown above the prompt.

```
SwiftOS CLI client v2.0

Connected to yourswiftcache (192.168.2.21)
Type 'help' if you need it
yourswiftcache>
add              get              raid             stats
brightcloud      help             reboot           supercluster
cache            idata            remove           test
cluster          interface        set              top100
config           keyhelp          shared           upgrade
edit             local            shell            upstream_policy
exit             no               show
filter           policy           shutdown
find             process          ssl
```

The following example uses tab completion first to list the possible process names then the command actions:

```
yourswiftcache> process
alertd          proxy           rtmp_proxy      taskmgr
intercept       rtmp_intercept  snmpagentd
yourswiftcache> process taskmgr
reload   restart  start    status   stop
yourswiftcache> process taskmgr status
taskmgr (pid  2975) is running...
```

9.3.3 CLI Help

The `find` command is used to locate configuration keys when an operator is unsure of the exact name of a configuration key. The `find` command takes a single argument, the text to search for. It will return the names of configuration keys, policies and filters that match the text.

For example, typing `find hostname` will return the `hostname` configuration key and show you that it is in the *local* section of SwiftCache's configuration:

```
yourswiftcache> find hostname
[local]
hostname = yourswiftcache
```

The `keyhelp` command can then be used to display a description of that key:

```
yourswiftcache> keyhelp hostname
System hostname
```

The `help` command describes how to use commands:

```
yourswiftcache> help find
Usage: find <regex>

search for '<regex>' in all sections
```

9.3.4 Checking Configuration

The `show` command displays the current value of a configuration key and takes the key name as the argument:

```
yourswiftcache> show hostname
yourswiftcache
```

To view the full device configuration, use the command `show config`.

Default Values

Almost every configuration key has a default value. The `show config` command only displays configuration keys that have been changed from their default values.

Configuration keys that retain their default setting are not displayed by `show config`. To view the complete device configuration, including the default values, use the command `show all_config`.

9.3.5 get and set Commands

All SwiftCache settings are presented as key and its corresponding value, e.g. `admin_port = 8500`. Configuration settings are viewed and changed using the `get` and `set` commands:

```
SwiftOS CLI client v2.0

Connected to nohostname (192.168.2.21)
Type 'help' if you need it
nohostname> get hostname
nohostname
nohostname> set hostname yourswiftcache
ok
nohostname> get hostname
yourswiftcache
```

9.3.6 add and remove Commands

The commands `add` and `remove` can be used to add and remove a value from a configuration list:

```
yourswiftcache> get ntp_servers
asia.pool.ntp.org
yourswiftcache> add ntp_servers local.ntp
ok
yourswiftcache> get ntp_servers
asia.pool.ntp.org,local.ntp
yourswiftcache> remove ntp_servers local.ntp
ok
yourswiftcache> get ntp_servers
asia.pool.ntp.org
```

9.3.7 Configuration Sections and Scope

SwiftCache's configuration is grouped into sections that control when the settings are applied, depending on the context or *scope*.

Configuration keys can appear in multiple sections, each with a different scope. For example, each configured *policy* section may define `cache_delay` with its own specific value.

9.3.8 create and no Commands

The `create` and `no` commands are used to create or delete a configuration value or section. The `create` command is used with a section name, for example:

```
yourswiftcache> create filter yourfiltername
ok
yourswiftcache> no filter yourfiltername
ok
```

filter Sections

Individual filter policies are each defined inside their own *filter* section in the configuration. Each named *filter* section contains the settings for that filter:

```
yourswiftcache> show filter yourfiltername

[filter-yourfiltername]
brightcloud_list = 01,02,03
default_deny = off
static_redirect_url = http://www.example.com/redirect/
whitelist_refresh = 0
whitelist_url = http://www.example.com/bypass.lst
```

Note that the section name is prefixed with `filter-` to show what type of configuration section is being displayed.

interface Sections

Each physical or virtual network interface on the SwiftCache has its own *interface* section in the configuration. Each named *interface* section contains the settings for that interface:

```
yourswiftcache> show interface eth0
[interface-eth0]
autoneg = off
dhcp = yes
duplex = full
enabled = on
```

policy Sections

Similarly, each policy has its own *policy* section in the configuration:

```
yourswiftcache> show policy dailymotion_1

[policy-dailymotion_1]
match url regex
http://.\*\\.dailymotion\\.com/(?<id\>video/\\d+/\\d+/\\d+[\\:|%3amp].\*)\\?.\*
cache_always = on
cache_index = dailymotion.com/$id
cache_ttl = 608400
pversion = 2
up_policy = dailymotion_1
```

upstream_policy Sections

upstream_policy sections look similar to *policy* sections, but are created from different sources. *upstream_policy* sections are created by SwiftCache after being downloaded automatically from SwiftSense.

policy sections contain policies that were created or edited locally. Upstream policies are for internal SwiftCache use only and should not be modified by operators.

shared Section

The *shared* section contains the configuration sections and keys that are shared across the cluster. Shared configuration settings are copied to other SwiftCaches by the cluster synchronisation process.

local Section

In contrast with the *shared* section, the *local* section defines the configuration keys and sections that override the shared cluster values and apply only to this SwiftCache appliance.

For example, `hostname` is a configuration key specific to an appliance and may only appear within the *local* section.

For more information on shared and local settings, please refer to **Cluster Settings** in the **SwiftCache GUI** chapter in this manual.

9.4 Other CLI Commands

9.4.1 brightcloud

The command `brightcloud status` reports on the current status of the *Brightcloud* module. For more information on Brightcloud, please refer to the **Brightcloud (Dynamic URL Classification)** section in the **Filtering** chapter of this manual.

9.4.2 cache

The `cache` command allows the operator to determine if a particular object is known to the proxy, and if so, some information about the way it is stored and has been accessed.

It is also possible to remove objects from the cache manually if they specifically need to be made unavailable. In normal operation, old items are automatically purged from the cache over time.

The actions `info` and `remove`, followed by an URL, are used to view information on objects cached on this appliance and delete them. Note that long lines are wrapped in the example below.

```
yourswiftcache> cache info http://example.com/img.jpg
gzip_properties: {'index': 'example.com/img.jpg:CE:gzip', 'path': 'Not found'}
properties: {'index': 'example.com/img.jpg', 'Content-Length': '2124', 'Last-V
alidated': 'Tue 02/Oct/2012 16:21:40 (1349194900)', 'Created': 'Tue 02/Oct/201
2 16:21:40 (1349194900)', 'Last-Accessed': 'Tue 02/Oct/2012 16:21:40 (13491949
00)', 'Expires': 'Fri 22/Aug/2014 10:32:32 (1408703552)', 'Server': 'Apache',
'Last-Modified': 'Mon 06/Aug/2012 12:13:50 (1344255230)', 'location': 'memory'
, 'Cache-Control': 'max-age=59508652', 'path': 'N/A', 'Content-Type': 'image/j
peg'}
```

The actions `clusterinfo` and `clusterremove` do the same but for the whole cluster.

9.4.3 cluster

The `cluster` command is used to manage clusters. Available actions are:

`join`	Current machine will join cluster of specified machine.
`leave`	Current machine will exit the current cluster and no longer synchronise its configuration.
`show`	Displays current status of the cluster.
`sync`	Forces a synchronisation of the configuration within the cluster.

9.4.4 config

The `config` command is used to manage the appliance's configuration. Available actions are:

`backup`	Create a backup of this configuration.
`clone`	Clones the configuration of the specified appliance to this appliance.
`delete_backup`	Removes a stored backup.
`diff`	Shows the differences between the active configuration and the specified backup.
`list`	Lists all backups.
`restore`	Reverts configuration to the specified backup.
`set`	Enables the batch insertion of raw configuration data. This is ideal for cutting and pasting sections from another machine. Exit this mode with `[Ctrl-d]`.
`show`	Displays the active raw configuration.

9.4.5 edit

The `edit` command allows modification of configuration keys using a text editor within the CLI. When the file is saved the configuration is updated.

If the file is discarded without saving then the configuration will remain unchanged.

9.4.6 exit

The `exit` command exits the CLI and ends the SSH, console or serial login session. The keyboard shortcut `[Ctrl]-d` may also be used.

9.4.7 idata

The `idata` command allows the operator to perform `get`, `set` and `delete` actions on internal data values. The action `list` will enumerate all the variables.

Great care should be taken since as incorrect use of the `idata` command can permanently disable the SwiftCache appliance and invalidate your warranty.

9.4.8 policy

This command manages *policy* configuration sections. It has the following actions:

`create <policyname>`	Create a policy called `<policyname>`.
`delete <policyname>`	Delete a policy.
`edit <policyname>`	Edit a policy.
`list`	List all policies.
`<policy_a> move before <policy_b>` `<policy_a> move after <policy_b>`	Reorder a policy.
`show <policyname>`	Show a policy.
`showall`	Show all policies.

More detail is available in the **Policies** chapter of this manual.

9.4.9 process

The `process` command manages the processes on the SwiftCache. The arguments for the command are the name of the process followed by an action.

The names of the processes are:

`proxy`	The main SwiftCache proxy process that handles HTTP traffic.
`intercept`	Intercepts HTTP traffic on port 80 and passes it to the SwiftCache proxy process.
`taskmgr`	Handles log rotation and other admin processes.
`alertd`	Handles creation and sending of alerts.
`snmpagentd`	Provides SNMP information to external monitoring systems.
`rtmp_proxy`	SwiftCache RTMP proxy process for handling RTMP traffic.
`rtmp_intercept`	Intercepts RTMP traffic on port 1935 and passes it to the RTMP proxy process.

The available actions are:

`reload`	Causes the process to reload its configuration.
`restart`	Restarts the process.
`start`	Starts the process if stopped.
`status`	Displays process status information.
`stop`	Stop the process if started.

The following example uses tab completion (pressing `[Tab]` twice) first to list the possible process names then the command actions:

```
yourswiftcache> process
alertd          proxy           rtmp_proxy      taskmgr
intercept       rtmp_intercept  snmpagentd
yourswiftcache> process taskmgr
reload   restart  start    status   stop
yourswiftcache> process taskmgr status
taskmgr (pid  2975) is running...
```

Care should be taken to avoid causing any disruption to client connections to the SwiftCache appliance. Remove traffic from the SwiftCache before stopping any processes.

9.4.10 raid show

The `raid` command takes a single action, `show`, and displays the RAID configuration of the SwiftCache appliance's storage media.

The available parameters are `controllers`, `volumes` and `disks`. For example:

```
yourswiftcache> raid show controllers
```

9.4.11 reboot and shutdown

The `reboot` command will restart the SwiftCache appliance. The `shutdown` command will shut down and power off the SwiftCache appliance.

Care should be taken to avoid causing any disruption to client connections to the SwiftCache appliance. Remove traffic from the SwiftCache before initiating an appliance reboot or shutdown.

9.4.12 shell

SwiftCache is built upon a Linux base operating system. The `shell` command allows the operator to initiate an interactive *shell* on the appliance.

The `shell` command allows suitably-experienced operators to run native Linux commands for advanced troubleshooting and diagnostics. Under normal circumstances it should not be necessary to use this command.

Great care should be taken since as incorrect use of the shell can permanently disable the SwiftCache appliance and invalidate your warranty.

9.4.13 ssl

The `ssl` command can be used to generate a self-signed SSL certificate to secure connections to the web-based GUI. This is required if the `admin_ssl` key is enabled in the configuration settings.

The command is also used to manage the SSL certificates used by the optional SSL caching module, if it is installed.

The available actions are:

`edit`	Edits an SSL certificate.
`keygen`	Generates SSL keys.
`list`	Lists available SSL certificates.
`show`	Shows detail of a specific SSL certificate.

9.4.14 stats

The `stats` command displays real-time statistics on a page that refreshes periodically. Press the `[Esc]` key to return to the command prompt.

```
Stats as of 2013-03-26 15:23:00 [Hit Esc to exit]
                                                    current
============================================================
                          avg_hit_time            78.0
                          avg_hit_ttfb            78.0
                         avg_miss_time            42.0
                         avg_miss_ttfb            42.0
                              avg_ttfb            42.0
                        byte_hit_rate             0.0
                     cached_bytes_rate            0.0
                             client_in            0.0
                    client_in_overload            0.0
                            client_out            0.0
                   client_out_overload            0.0
                              con_rate            0.0
                                   cpu            5.0
                              cpu_idle           95.0
                               cpu_iow            0.0
                               cpu_irq            0.3
                            cpu_system            1.3
                              cpu_user            3.5
                        disk_sda_atime            0.0
                        disk_sda_await            0.8
                          disk_sda_rps            0.0
                         disk_sda_util            0.2
                          disk_sda_wps            5.1
                      disks_cache_await            0.0
                       disks_cache_util            0.0
                       disks_fast_await            0.0
                        disks_fast_util            0.0
                      disks_other_await            0.0
                       disks_other_util            0.0
                              disks_rps            0.0
```

9.4.15 supercluster

This command is used to manage *superclusters*. Available actions are:

`join`	Current machine will join supercluster of specified machine.
`leave`	Current machine will exit the current supercluster.
`show`	Displays current status of the supercluster.

For more information on superclusters, please refer to the **Clustering** chapter in this manual.

9.4.16 test

The command `test log_upload` will attempt a log upload using the current settings.

9.4.17 top100

The `top100` command reports on the most commonly active clients and/or sites. This allows an operator to determine the most heavy users and frequently-accessed sites. This information can then guide the operator when tuning the SwiftCache configuration and policies for optimal performance.

For example:

```
yourswiftcache> top100 <timeframe> <objects> <units>
```

where:

- `<timeframe>` is replaced with `hourly`, `daily`, `monthly`, `weekly` or `yearly`;
- `<objects>` is replaced with `clients`, `sites` or `videos`; and
- `<units>` is replaced with `bytes` or `requests`.

9.4.18 upgrade

This command is used to perform a software upgrade in a cluster. It has the following actions:

`delete_version <version>`	Delete specified cache `<version>` from the database.
`list_versions`	List known cache versions.
`log`	Show upgrade log.
`perform <version>`	Perform upgrade to specified cache `<version>`.

`prepare <version>`
`prepare <rpm url>`
`prepare <filename>`
	Prepare upgrade using specified `<version>`, `<rpm url>` or `<filename>`.

`status`	Show upgrade status.

More detail on the cluster upgrade process is available in the **Clustering** chapter of this manual.

9.4.19 upstream_policy

This command manages *upstream_policy* configuration sections. It has the following actions:

`info <policyname>`	Info about an upstream policy.
`install <policyname>`	Install an upstream policy.
`install_all`	Install all upstream policies.
`list`	List all upstream policies.
`show <policyname>`	Show an upstream policy.
`uninstall <policyname>`	Uninstall an upstream policy.
`uninstall_all`	Uninstall all upstream policies.
`upgrade <policyname>`	Upgrade an upstream policy.
`upgrade_all`	Upgrade all upstream policies.

More detail is available in the **Policies** chapter of this manual.

10 Operations Guide

This chapter provides guidance on common operational tasks.

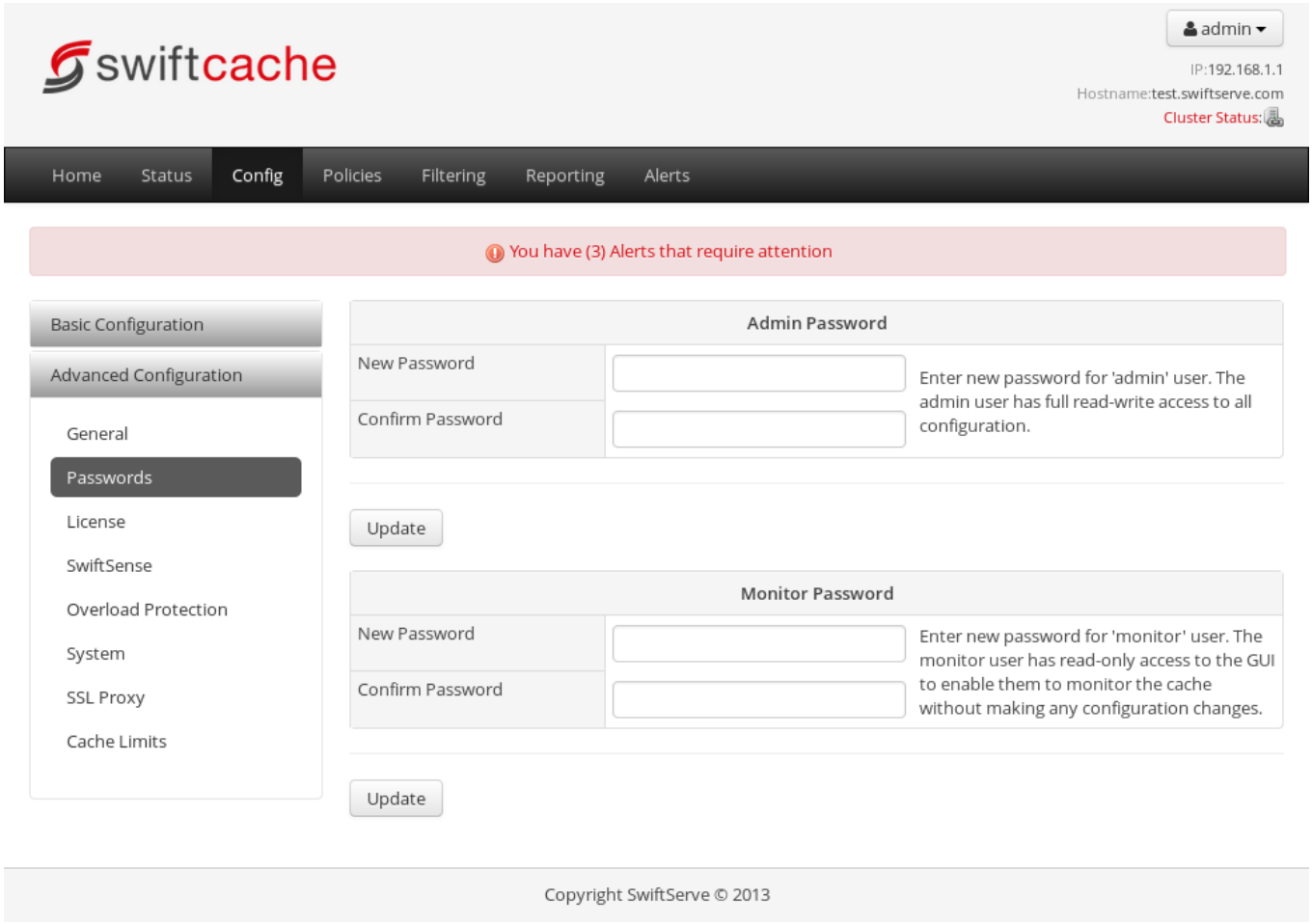
10.1 Securing SwiftCache

Access to the SwiftCache GUI and CLI are protected by username and password. Additionally communications with the web interface may be SSL-encrypted. To enable this run the command `set admin_ssl on` in the CLI:

```
yourswiftcache> set admin_ssl on
ok
```

It is additionally recommended that an external security device or access control list is applied to the GUI to prevent access from unauthorised or external users.

10.1.1 User Administration



Two users exist within the SwiftCache GUI:

- The `admin` user has full read/write access to the interface and can modify any settings.
- The `monitor` user has read-only access to the Dashboard, Reporting and Alerts pages.

The `admin` user is also the only account that can access the CLI.

Passwords for both users are maintained using the *Passwords* section under *Advanced Configuration* in the *Config* tab in the GUI.

10.1.2 Best Practice Recommendations

- Enable SSL on the GUI.
- Change the user passwords after installation, when any operators leave and at least once per year.
- Disable sending of `X-Cache-Debug` HTTP headers as these reveal debugging information about the cache.
- Disable `allow_explicit` (if specifically required, allow only via a policy).
- Enable `always_do_dns`.

10.2 Disk Replacement

The procedure to replace the hard disks in the SwiftCache appliance will vary depending on the hardware specification. Please contact the SwiftCache Support Team for guidance on disk replacement.

10.3 Configuration Backup

10.3.1 GUI

The *Backups* portion of the *Basic Configuration* subsection of the *Config* tab in the GUI allows an operator to create a backup of the current configuration as well as delete backups, view differences between backups and restore previous backups.

10.3.2 CLI

Running `show config` will dump the running configuration as text that can be exported for archiving elsewhere.

11 Monitoring

This chapter discusses the different ways in which the SwiftCache appliance provides information on its current and previous status and performance, and raise alerts in certain circumstances.

11.1 Reporting

There are three different ways to retrieve monitoring information from a SwiftCache. One or more options may be applicable in a deployment scenario depending on the relative ease of integrating each one with existing tools and practices.

Web Interface	The *Reporting* tab of the SwiftCache GUI provides extensive information on the local and cluster performance and health.
Command Line Interface	The CLI provides a `stats` command that allows the low-level performance of the local machine to be analysed, and also top 100 information.
SNMP	All aspects of the appliance to be queried remotely from an existing network monitoring system.

11.1.1 GUI

Please refer to the **SwiftCache GUI** chapter in this manual for details of the information available on the *Reporting* tab.

11.1.2 CLI

Please refer to the **SwiftCache CLI** chapter in this manual for details of the information available from the `stats` and `top100` commands.

11.1.3 SNMP

SwiftCache provides an extensive Management Information Base (MIB) that allows allows monitoring of all key statistics via SNMP; please refer to **Appendix B** for details of the SwiftCache v2.x MIB.

SwiftCache uses SNMPv2c community-based authentication. The default community string is `public`. This can be modified on the *General* portion of the *Advanced* subsection of the *Config* tab in the GUI, or by changing the `snmp_community` configuration key using the CLI.

11.2 Alerts

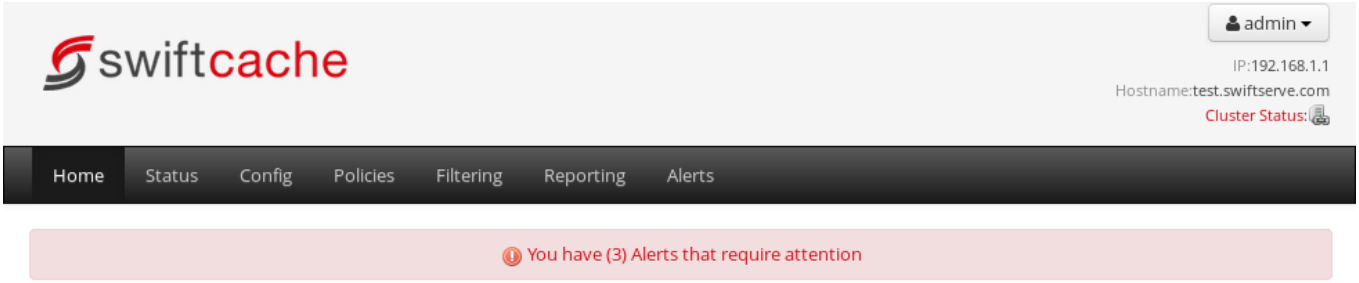
11.2.1 Alert Framework

The SwiftCache alert framework proactively monitors system health and identifies issues that require attention or acknowledgement. This means that the operator does not need to identify unusual behaviour by manually monitoring multiple log files, system messages and performance statistics.

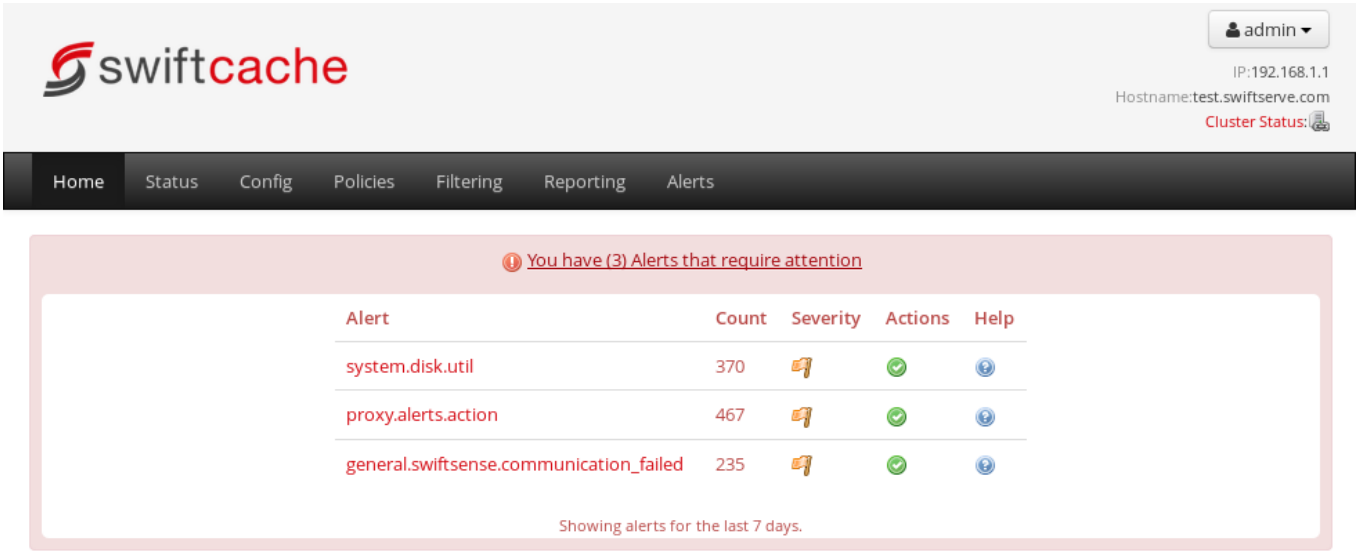
The alert framework provides a consolidated view of all issues that require operator intervention on the local machine or across the entire cluster. The framework provides a prioritised view of all alerts and collects similar issues together. This avoids the operator from being overwhelmed by an *alarm flood* where alerts are reported many times for the same issue.

Issues that require operator attention or action can be notified. SwiftCache supports multiple notification mechanisms that can easily be integrated into an existing network management system: email, SNMP trap, HTTP web service and Syslog. For more information, please see the **Alert Notifications** section below.

To ensure that operators are aware of alerts, the GUI presents a count of the number of unacknowledged alerts within the alert bar visible on all tabs:



Clicking on this alert bar expands the summary of the outstanding alerts:

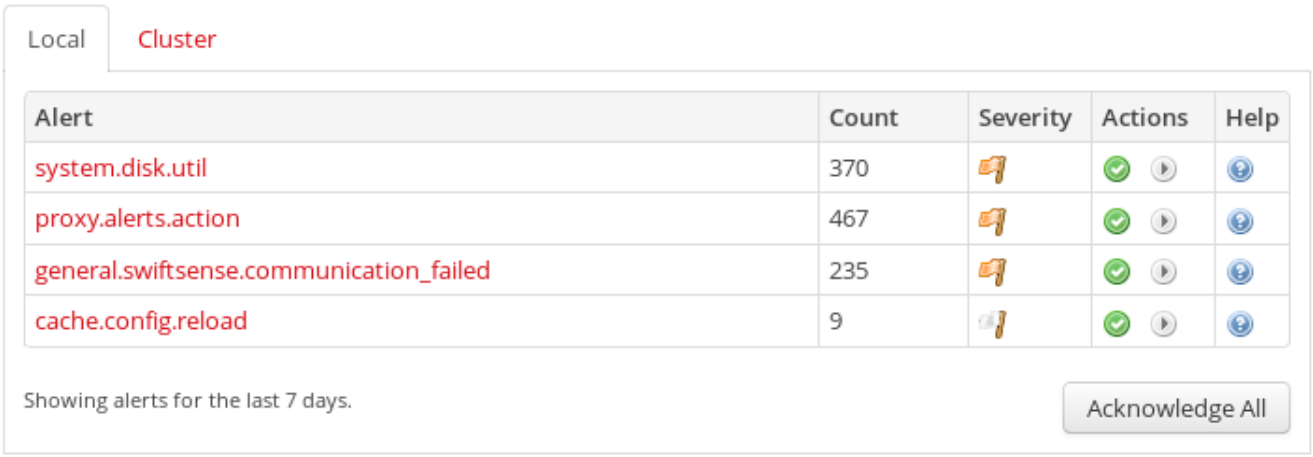


11.2.2 Alert Workflow

When new alerts are raised they may optionally trigger a notification. New alerts are also added to the count of unacknowledged alerts in the header bar.

Alerts will remain in an unacknowledged state until an operator manually acknowledges the alert. Acknowledging an alert removes it from the alert bar and the *Alerts* tab in the GUI. Acknowledged alerts are hidden by default but can be viewed by changing the settings on the alert page.

An operator may acknowledge an alert directly from the alert bar, or act upon it by going to the *Alerts* tab in the main navigation bar. To acknowledge all occurrences of an alert, an operator can click on the green tick in the *Actions* column:



If an alert is expanded to show the details then each instance of an alert may be acknowledged separately.

Acknowledged alerts are deleted from the SwiftCache after seven days.

11.2.3 Alert Suppression

To ensure that SwiftCache does not flood the operator with issues, SwiftCache manages the information that is presented to the operator in two ways:

- Alerts of exactly the same type are suppressed for ten minutes. This means that if a condition such as high CPU usage persists for a long duration, an alert will only be generated once every ten minutes. This removes redundant information from the interface that would simply confirm that a known condition still exists.
- Alerts of the same type are grouped in the interface and a count of the instances of each type presented. This grouping means that issues that are *likely* to have the same cause are listed as a single entry in the interface. This also presents a more compact view of the alerts, allowing an operator to quickly identify important and urgent alerts without being swamped by noise.

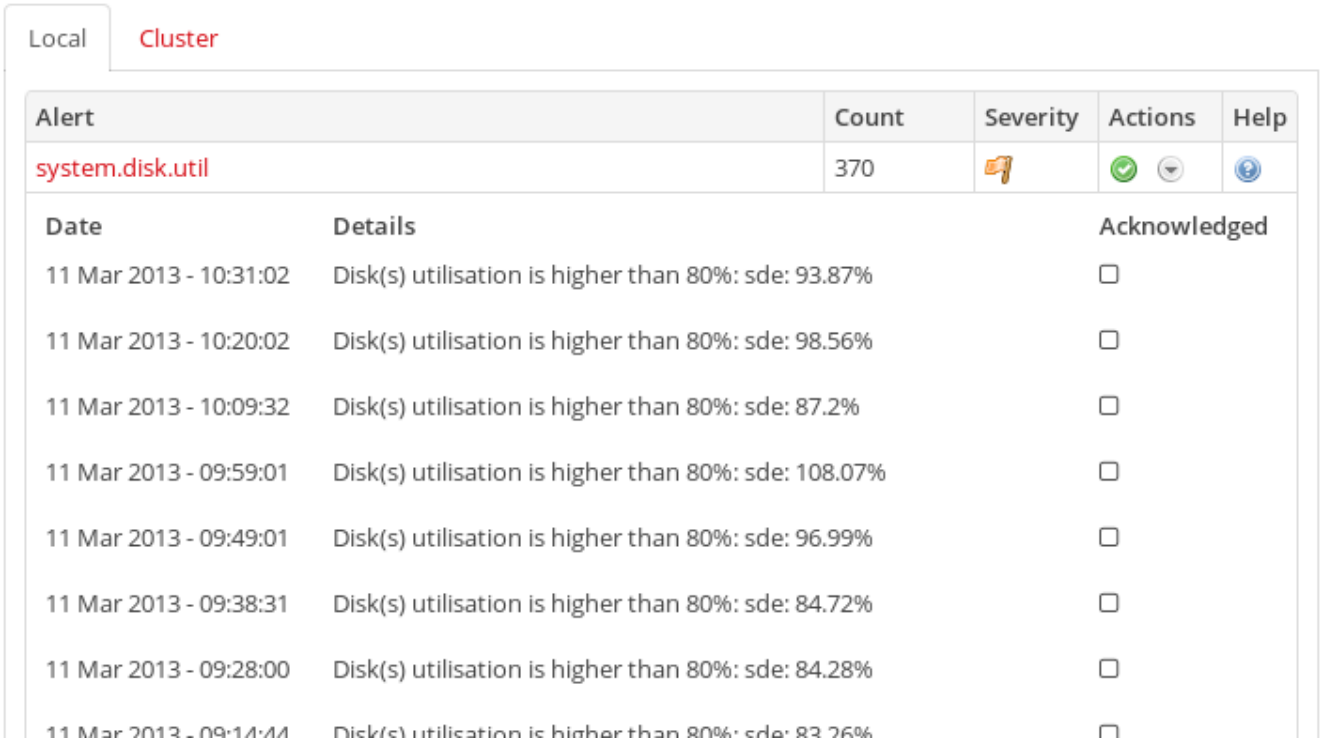
11.2.4 Alert Priorities

Each alert is assigned a priority which corresponds to the severity of that alert. The different alert priorities are represented in the GUI with differently coloured flags. Five alert severities are defined:

	1 (Trivial)	Lowest severity level. These alerts are designed to inform a user but do not require any response. For example `cache.config.reload` informs the user that the configuration was reloaded after applying a configuration change.
	2 (Minor)	These alerts raise attention to a non-critical issue that might lead to unexpected behaviour if they persist for some time. For example `general.log.files_deleted` informs the operator that the log manager has deleted old log files based upon log retention policies.
	3 (Warning)	These alerts indicate an issue that should be investigated to determine the root cause. They do not constitute a critical or serious issue by themselves, but they may be indicative of an underlying and more serious problem. For example `system.cpu.high_usage` indicates that the SwiftCache is reaching the limit of CPU resources and may trigger Overload Bypass mode (depending on thresholds).
	4 (Serious)	These alerts indicate an issue that should be investigated immediately as it could be degrading performance or impairing operation of the SwiftCache. For example `system.net.config.update_failed` indicates that there was an error applying new configuration the network interfaces and the old config has been restored.
	5 (Critical)	Highest severity level. These alerts indicate that the performance of the SwiftCache is being degraded and immediate action should be taken to restore the service. For example `system.overload` indicates that the load threshold as been breached and the system has triggered the Overload Protection mechanism to place new connections into bypass or relay modes.

11.2.5 Alert Details

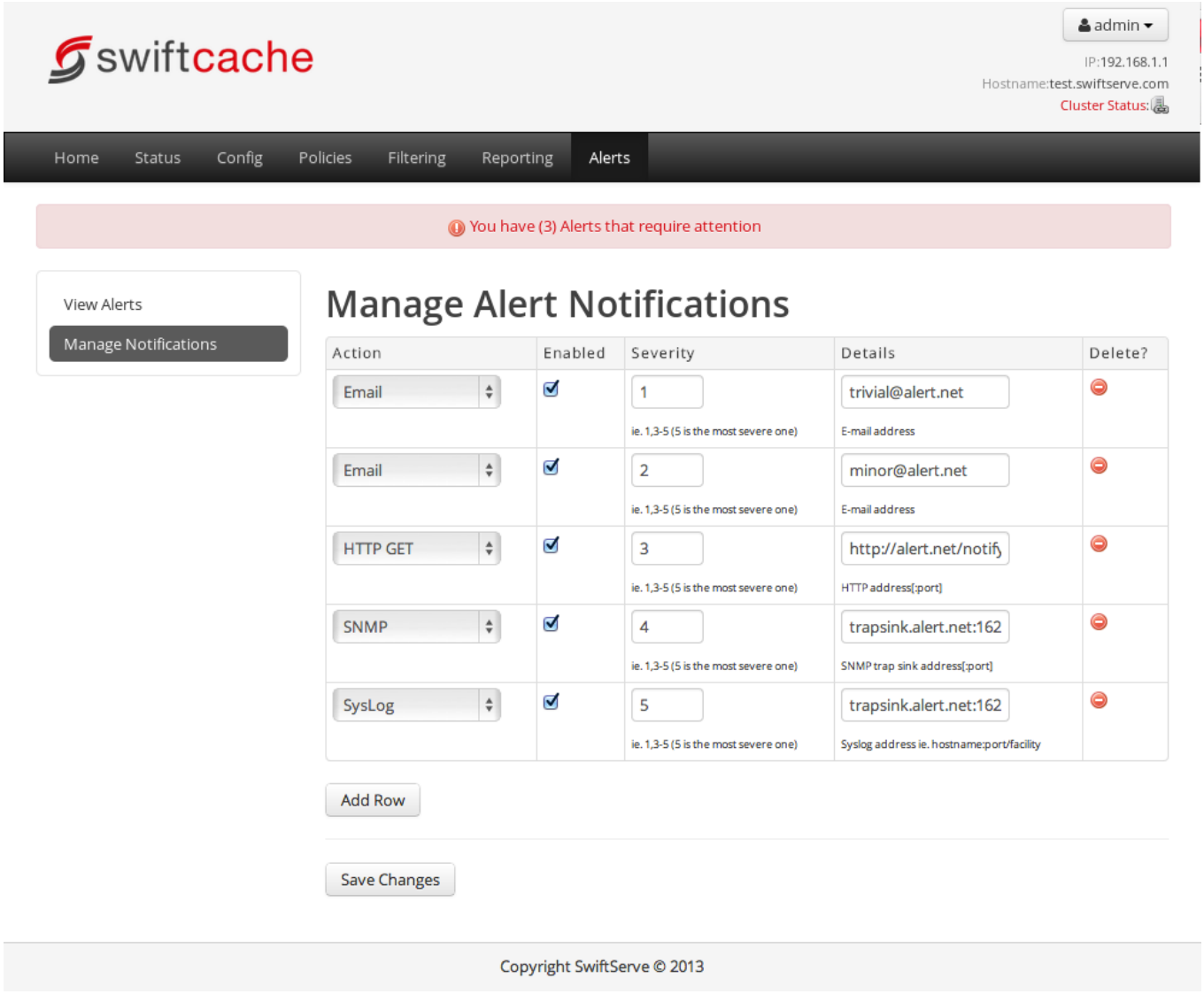
Each alert instance contains a specific description of the conditions that gave rise to that alert. This information can be especially valuable in diagnosing the cause of the issue and how to resolve. To view the details for each alert instance select the arrow icon in the *Actions* column.


Local
Cluster
Alert
Count
Severity
Actions
Help
system.disk.util
370
Date
Details
Acknowledged
11 Mar 2013 - 10:31:02
Disk(s) utilisation is higher than 80%: sde: 93.87%
11 Mar 2013 - 10:20:02
Disk(s) utilisation is higher than 80%: sde: 98.56%
11 Mar 2013 - 10:09:32
Disk(s) utilisation is higher than 80%: sde: 87.2%
11 Mar 2013 - 09:59:01
Disk(s) utilisation is higher than 80%: sde: 108.07%
11 Mar 2013 - 09:49:01
Disk(s) utilisation is higher than 80%: sde: 96.99%
11 Mar 2013 - 09:38:31
Disk(s) utilisation is higher than 80%: sde: 84.72%
11 Mar 2013 - 09:28:00
Disk(s) utilisation is higher than 80%: sde: 84.28%

In the screenshot above the `general.swiftsense.misconfiguration` alert has been raised ten times. By expanding the alert details, the operator is able to see that each of these errors was triggered by a failure to connect to the SwiftSense update server.

The alert details also provide the exact connection error in each case and a timestamp for when the error occurred to allow appropriate log files to be pinpointed.

11.2.6 Alert Notifications



SwiftCache supports four types of notification that may be raised for any alert instance:

Email	An email will be sent to the specified email address with the alert instance details. For emails to be sent, a valid SMTP relay must be defined with the following configuration keys: • `smtp_host` • `smtp_port` • `smtp_username` • `smtp_password` and • `smtp_ssl`. See **Appendix C** for more details of these configuration keys.
SNMP Trap	An SNMP Trap will be sent to the specified sink address with the alert details encoded into the trap payload. For more details of trap OIDs, please refer to **Appendix B**.
HTTP Web Service	An HTTP GET request will be made to the specified URL with the following query parameters: • `Alert`: The alert type, e.g. `general.swiftsense.misconfiguration`. • `Timestamp`: The alert timestamp in `ddmmyy-HHMMSS` format. • `Server`: The IP address of the cache generating the alert. • `Arg`: The alert argument containing the alert details.
Syslog	SwiftCache can be configured to report key error messages to a central syslog server that can be monitored by the network operator.

Alert notifications are configured in the GUI from the *Manage Notifications* subsection of the *Alerts* tab. One or more actions may be configured for each severity of alert.

12 Performance Tuning

One of the key challenges for any web caching solution is ensuring that it is operating at maximum efficiency. This is especially difficult when caching Internet traffic. This is because the exact load on each cache will be very specific to the traffic pattern that the SwiftCache experiences.

By default SwiftCache uses a general-purpose performance configuration that will cope with the majority of deployments. However it may be possible to improve the performance of SwiftCache further by tuning the configuration for the specific traffic load profile.

12.1 Factors Affecting Performance

To maximise performance, it is necessary to understand the main factors affecting SwiftCache performance. The three key resources that SwiftCache relies upon are *CPU*, *memory* and *disk IO*.

The rate at which these resources are consumed will vary based on the traffic load that the SwiftCache appliance experiences. In many cases the maximum traffic load that can be achieved in any individual deployment will depend upon all the following factors.

12.1.1 Request Rate

Request rate is one of the more reliable indicators of load on a cache. For a given traffic mix, it is reasonable to assume that the CPU utilisation will vary linearly with request rate. For example, if the average CPU utilisation across all cores is 20 percent at 1000 requests per second (req/sec), it will be approximately 40 percent at 2000 req/sec.

12.1.2 Simultaneous Connections

Each TCP connection that the cache has open consumes a certain amount of memory to maintain state. The underlying operating system of the appliance also has limitations on the number of ports that are available.

12.1.3 Cache Efficiency

The average service time for cache hits is much lower than cache misses as the SwiftCache can avoid the round trip to the origin server. This means that the SwiftCache is able to complete the requests resulting in cache hits more quickly. When the cache is operating more efficiently, the number of simultaneous connections will be lower for a given request rate.

12.1.4 Client Speed

In a 2.5G mobile network the average client connection speed may be less than 100 kilobits per second (kbps). This contrasts with a fixed network, where the average client throughput is typically greater than 1 megabit per second (Mbps). Slower client connection speeds result in a much longer service time for a given traffic pattern. This longer service time will result in more simultaneous connections in the mobile network than for the same request rate as a fixed network.

12.1.5 Filtering

If filtering rules are enabled on the SwiftCache this will consume additional resources to apply the filtering chain to matching connections. Hence for a given request rate, CPU usage will be higher if filtering is enabled, depending on the complexity of the filtering chain.

12.1.6 Policies

Policies are applied to each incoming connection. As the number of policies and the complexity of their match

rules increases, the CPU usage will correspondingly increase for the same request rate.

12.1.7 Disk IO

The single biggest bottleneck that affects performance in typical caching scenarios is disk IO (input/output). Disks in caches are subject to many random operations and non-sequential reads. For a typical cache disk under full load, up to 80 percent of the disk time is spent seeking, i.e. moving the disk head to read a new file.

The performance of the disk will be substantially worse if the average file size is small. This is because the ratio of disk seek to read/write time is much higher than with large files, from which more data can read sequentially between disk seeks.

12.2 Improving Performance

The ideal traffic pattern for optimal SwiftCache performance is requests with a high cache hit-rate, for large files, from fast clients. For example, delivering Windows Update traffic would be an ideal traffic profile for SwiftCache to improve through caching of content.

Conversely, traffic consisting of requests with a low cache hit-rate, for small files, from slow clients is harder to improve through caching. For example, delivering highly dynamic, personalised AJAX websites to mobile clients.

The following steps are taken by the SwiftCache to optimise performance:

- An in-memory cache is utilised for small objects to avoid the time penalty of disk access.
- Disk access is avoided for *long-tail* content until SwiftCache has identified that the content is popular. This avoids wasting resources by storing content that will not be requested frequently and is controlled by the `cache_delay` configuration key.
- Disk bypass is enabled for sites that are identified as uncacheable. This prevents SwiftCache from wasting disk resources on content that will not result in a cache hit and is controlled by the `cache_never` configuration key.

12.2.1 Maximising Bandwidth Savings

Many Internet websites are not designed to take full advantage of the caching techniques available in the HTTP specification. For example, many sites make use of query string parameters to pass details of the requested content:

```
http://v18.lscache7.c.youtube.com/videoplayback?ip=0.0.0.0&sparams=id%2Cexpire%2Cip%2Cipbits%2Citag
%2Cburst
%2Cfactor&itag=34&ipbits=0&signature=8B482CBEDB4DAF3E9E044A4C25DFC881B8668E4F.BC606865625583C2BE5AA6472A7536D827ED7E58&sver=3&expire=1252105200&key=yt1&factor=1.25&burst=40&id=c19764decede17ab

http://v10.lscache5.c.youtube.com/videoplayback?ip=0.0.0.0&sparams=id%2Cexpire%2Cip%2Cipbits%2Citag
%2Cburst
%2Cfactor&itag=34&ipbits=0&signature=98803AA4E65D830E33C9A2FBC2FE77D02015257F29879DD9541B65704625CAB118092DFD151856CF&sver=3&expire=1252108800&key=yt1&factor=1.25&burst=40&id=c19764decede17ab
```

These types of URL are semi-dynamic. There is a static portion of the URL that is used to identify the requested file. The query string of the URL will vary dynamically with each request. SwiftCache provides a flexible policy mechanism to manipulate request URLs to extract the relevant information to allow more effective caching.

In some cases, websites will use fully-dynamic URLs. In the example below, each request is for the same file but the URL cannot reliably be used to determine this:

http://www846.megavideo.com/files/d008f8c759a4f4b3f07ccef7ea7588a4/

http://www763.megavideo.com/files/f66f936f7fc39f1bb9524f49c5f54184/

To cope with such cases SwiftCache provides a *content hashing* feature. This inspects the content payload without reference to the URL to identify when the content requested will benefit from caching.

SwiftSense

One of the benefits of deploying a SwiftCache solution is use of SwiftSense. SwiftSense is a cloud service that continually analyses millions of requests passing through SwiftCaches around the world. It uses this information to determine the optimal caching policies. SwiftSense then automatically provides tailored policy updates to live SwiftCaches. Please refer to the **Policies** and **SwiftSense** chapters in this manual for more detail.

13 Clustering

The management of many servers in a large infrastructure deployment can be time-consuming and prone to human error. To solve this problem, multiple SwiftCache appliances can be configured and managed as a single entity, or *cluster*.

13.1 Overview

SwiftCache's cluster management features allow operators to:

- propagate configuration changes across an entire cluster with a single action (please refer to the **Cluster Configuration** section below);
- review the performance of the entire cluster through a single interface (please refer to the **Reporting** section in the **SwiftCache GUI** chapter of this manual);
- run SwiftCache appliances with no single points of failure in a master/master architecture, enabling maximum scalability and stability by avoiding split network or consistency issues;
- define superclusters to aggregate reporting data from multiple clusters, each with their own distinct configurations (see the **Superclusters** section below);
- upgrade appliances in a cluster sequentially (please refer to the **Automated Cluster Upgrades** section later in this chapter); and
- make additional performance, bandwidth and disk space savings through **Inter-Cache Communication (ICC)** described later in this chapter.

13.2 Cluster Configuration

SwiftCache includes sophisticated tools to manage cluster configuration and synchronisation. Configuration changes can be propagated across an entire cluster with a single action. It is also possible to review the configuration across all members (*nodes*) of the cluster to identify anomalous configuration and quickly reconcile those differences.

The majority of configuration changes can be made without requiring any system downtime or the need to remove traffic from the cluster. Only low-level changes such as modifying network settings may require a restart and therefore should be scheduled during a maintenance window.

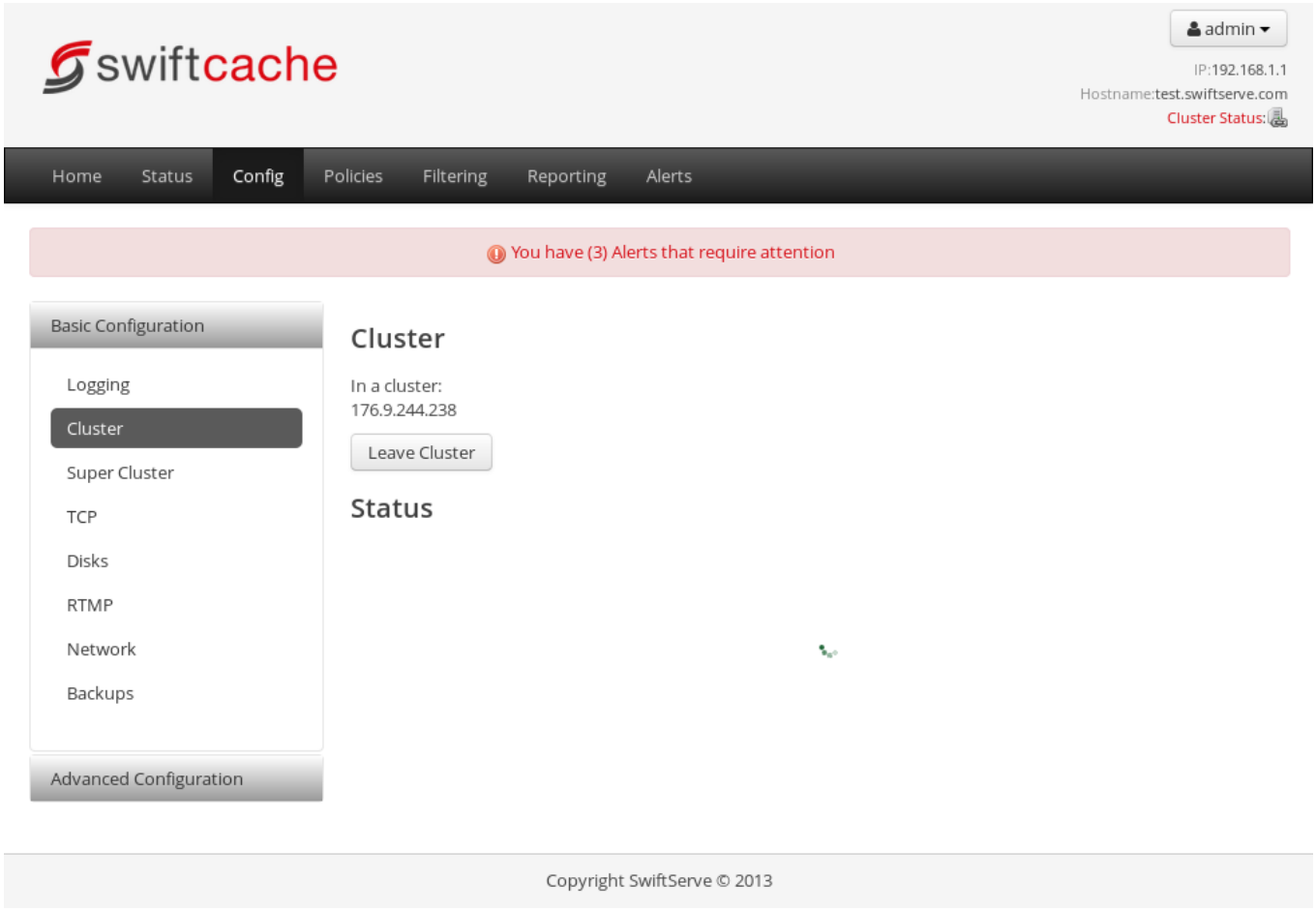
Configuration management is also resilient to node failures. For example, configuration changes may have been made to the cluster while a node was powered-off or uncontactable. SwiftCache compares the configuration of all machines in the cluster and uses an intelligent voting mechanism (based on age and popularity) to determine the optimal configuration to be applied to nodes rejoining the cluster.

For more information, please refer to the **Cluster Settings** section in the **SwiftServe GUI** chapter.

13.2.1 Adding a New Appliance to a Cluster

A common operation task is adding a new SwiftCache to an existing cluster. This will clone the shared configuration from that cluster to the new appliance. The new appliance then remains synchronised with any future updates to the configuration. Joining a machine to a cluster also grants permission to the other nodes to gather reporting statistics from this new cache and vice versa.

This is achieved via the GUI with the *Cluster* subsection of the *Config* tab:

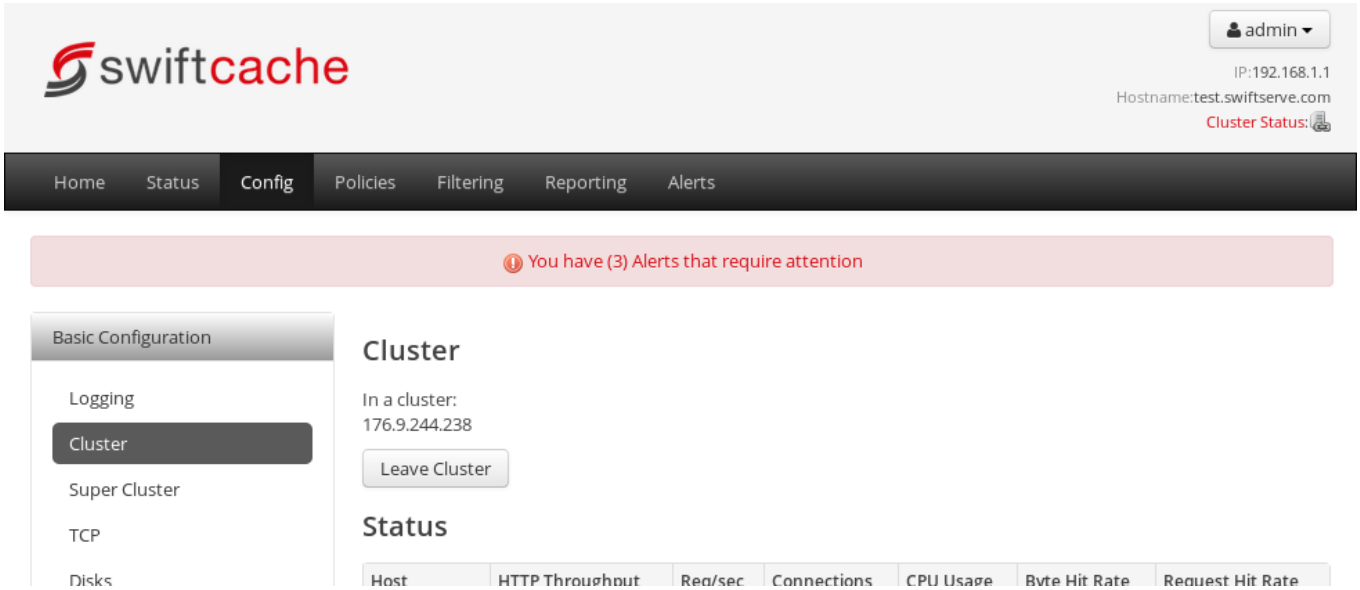


To join a new SwiftCache to a cluster, the operator needs to enter the IP address of any machine in the existing cluster. The operator also needs to provide an authentication credential to join that cluster. This can be either the `admin` user password or the secret passphrase stored in the `admin_secret` configuration key.

Once the new node has joined the cluster, the GUI will update to reflect that the machine is now a member of the cluster. The *Cluster* subsection of the *Config* tab will now show the IP addresses of the other cluster nodes.

13.2.2 Removing a Node from a Cluster

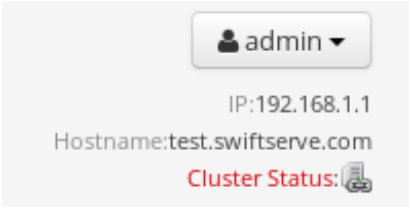
When a SwiftCache appliance is part of a cluster, the *Cluster* subsection of the *Config* tab in the GUI will show a button to leave that cluster. When the machine leaves a cluster its shared configuration remains “frozen” at the last synchronised state. Appliance configuration does not reset to default values on leaving a cluster.



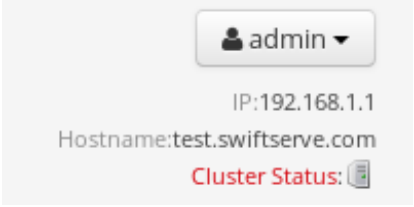
13.2.3 Viewing Cluster Status

The header of the GUI provides a visual representation of the current cluster status and health.

When a SwiftCache is synchronised with a cluster, an icon is displayed with a link of chain in front:



When a SwiftCache is not joined to a cluster the icon is shown without the link of chain in front.



From the CLI, running the `cluster` command shows the status of all cluster members and their synchronisation status. An MD5 hash of the configuration on each machine is used to verify that the machines are synchronised correctly:

```
yourswiftcache> cluster
Cluster Status
178.79.131.88
status: alive (2959 ms)
config: md5=029b6af57e5a7d44a937759b2efd8088 age=337
109.74.202.156
status: alive (35 ms)
config: md5=029b6af57e5a7d44a937759b2efd8088 age=330
best config is: 029b6af57e5a7d44a937759b2efd8088
our config is: 029b6af57e5a7d44a937759b2efd8088
```

13.3 Superclusters

A SwiftCache supercluster allows multiple SwiftCache clusters to communicate with each other to share reporting data. This can be useful where an organisation has multiple, distinct clusters on the network, each with its own unique configuration. This allows an operator to view how both clusters are performing from a single point.

For larger deployments, it is recommended that SwiftSense reporting is used in preference to using a supercluster when possible. For more detail, please refer to the **SwiftSense** chapter in this manual.

Superclusters are a reporting interface only. Configuration sharing, Inter-Cache Communication (ICC) and automated cluster upgrades do not operate in super clusters.

Supercluster reports are found within the GUI on the *Home (Dashboard)* and *Reporting* tabs.

13.3.1 Supercluster Setup

GUI

To create a supercluster from the GUI, two or more operational SwiftCache clusters are required. The operator needs to log in to any machine in one of the clusters. The operator then needs to enter the IP address of any machine in the other cluster and to provide an authentication credential to create the supercluster. This can be either the `admin` user password or the secret passphrase stored in the `admin_secret` configuration key for the other cluster. The supercluster is created by clicking the *Join Supercluster* button.

CLI

To create a supercluster from the CLI, the operator uses the `supercluster join` command. When prompted enter the IP address of the remote machine and the `admin` user password.

Operators can view the status of the supercluster in the CLI with the command `supercluster show`. To remove the current clusters from a supercluster, use the command `supercluster leave`.

13.4 Automated Cluster Upgrades

Automated cluster upgrades are performance using the CLI only. The process allows an operator to deploy a new SwiftCache software release across a cluster while providing information on the status of the upgrade. The upgrades are performed sequentially by SwiftCache, one node at a time, so that the performance of the cluster is maintained while the upgrade takes place.

An operator may also revert an upgrade (downgrade) the SwiftCache software to a known-good software release should a problem be encountered during the upgrade. SwiftCache will keep backup copies of the SwiftCache software only if installed by the automated upgrades feature. Manual software upgrades will not create a backup of the previous software or configuration.

13.4.1 Automated Upgrade Workflow

A *RPM* package file containing the new release of the SwiftCache software is needed to upgrade a cluster. To prepare the cluster for the upgrade, the operator needs to copy the RPM onto the SwiftCache itself or to a location where the file can be read from the SwiftCache (e.g. a web server). Once the upgrade has been performed, it is no longer necesssary to keep the RPM file as a backup copy will be stored automatically by SwiftCache.

In the CLI, the operator then issues the command `upgrade prepare` followed by the path or URL to the new SwiftCache RPM. The SwiftCache will then automatically distribute the RPM to all nodes in the cluster. To verify the RPM is now on all machines, use the command `upgrade list_versions`, which will show all available

versions of the SwiftCache software.

To perform the upgrade, use the command `upgrade perform <version>`, where `<version>` is the target version number of the software to upgrade to. Once the upgrade has been started, the current status of the upgrade can be viewed with the command `upgrade status`. For full details of the `upgrade` command please refer to the **SwiftCache CLI** section of this document.

Note that error messages may be generated while individual nodes in the cluster are upgraded. This is because each node will be unavailable for a brief period of time while the upgrade takes place.

When the upgrade is complete, the operator can verify when it was updated with the command `upgrade log`. Note that this command only shows the upgrade history for the local machine.

13.4.2 Failures During Upgrade

If the upgrade of an individual machine fails, no further upgrade action will be taken and the operator will have to intervene manually. When the issue has been resolved, the upgrade can be continued with automated upgrades by relaunching the process. If any machine has already been upgraded then there will be no further interruption of service on that machine.

It is recommended that to test correct operation of a SwiftCache cluster once an upgrade is complete. If a problem has occurs with the upgrade, it is possible to downgrade the cluster to a previous, known-good SwiftCache release.

13.4.3 Deleting Old Software Versions

To delete old versions from the list of available software for the cluster, issue the command `upgrade delete_version <old-version>` to remove the `<old-version>`. It is recommended that operators do not delete old software versions until after verifying the successful upgrade and correct operation of the SwiftCache cluster.

13.4.4 Important Warnings

When upgrading between SwiftCache software versions, obsolete configuration keys are deleted from the live settings. This may also occur when downgrading a SwiftCache if a configuration key that is present in a newer software version is not present in the previous version.

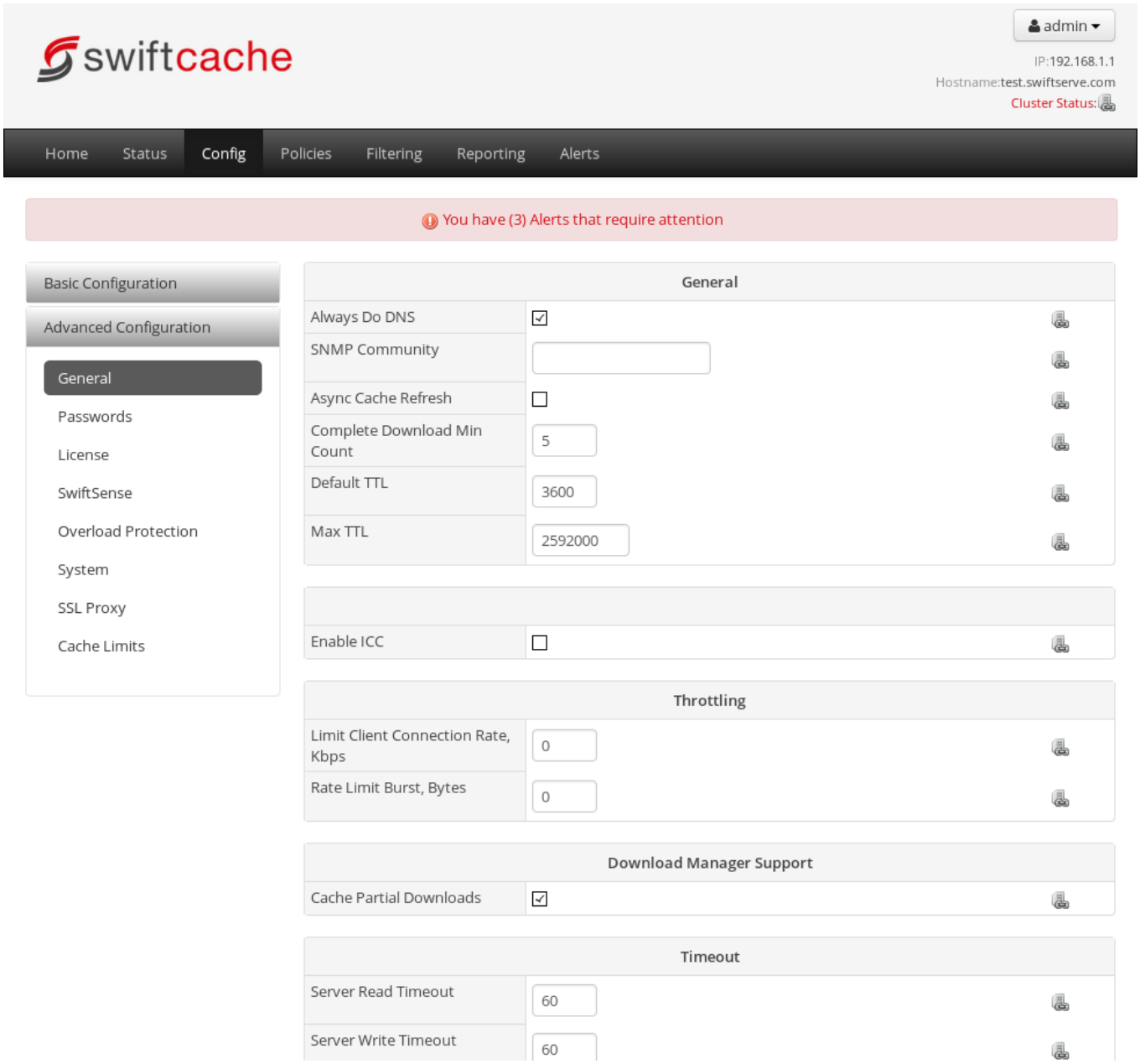
This may result in a situation when downgrading that a configuration may not be return to exactly the same state as it was before the failed upgrade attempt.

It is recommended to take a manual backup of the configuration before starting an upgrade. The configuration is stored on an appliance `/etc/default/`.

13.5 Inter-Cache Communication (ICC)

Inter-Cache Communication (ICC) improves cache hit rates and delivers bandwidth savings. It does this by allowing nodes to share and distribute cached objects around the SwiftCache cluster without each machine having to retrieve content from the origin servers. ICC also makes best use of the available disk space in a cluster.

13.5.1 ICC Setup



ICC is enabled via the GUI in the *General* portion of the *Advanced Configuration* subsection of the *Config* tab by ticking the *Enable ICC* box then clicking *Update*.

In the CLI, use the command `set icc_enabled on`. To disable ICC use the command `set icc_enabled off`.

13.5.2 Request Workflow Without ICC

Without ICC, bandwidth and storage resources are used inefficiently. If ICC is disabled in the SwiftCache cluster:

- When a SwiftCache receives a client request for an object for the first time, it requests the item from the origin server and returns it to the client.
- Subsequent client requests for the same object may go to other SwiftCaches in the cluster.
- As the other nodes have not yet cached this object, each SwiftCache would independently request and cache the object.

This results in multiple additional requests to the origin server from the SwiftCache cluster. Each node will also store its own copy of the cached object on disk.

13.5.3 Request Workflow With ICC

With ICC, bandwidth and storage resources are conserved. If ICC is enabled in the SwiftCache cluster:

- As before, when a SwiftCache receives a client request for an object for the first time, it requests the item from the origin server and returns it to the client.
- SwiftCache then adds an entry to the cache database shared by the cluster.
- This identifies the item uniquely by hash and designates which specific node will be responsible for storing the item on disk.
- Subsequent client requests for the same object may go to any node in the cluster.
- If the request goes to the one node now responsible for storing the object, it returns the object to the client.
- If the request goes to a different cache, that cache retrieves the object from the node storing it (rather than the origin server), then returns it to the client.

This means that the SwiftCache cluster does not have to make multiple requests for the same object to the origin server, saving bandwidth. It also means that the cluster only has to keep one copy of the object, saving disk space.

13.5.4 Advanced Information

Items Not Handled by ICC

Certain content items will not be handled by ICC:

- items smaller than `icc_min_size` (by default 16kB);
- items that cannot be cached; and
- items that have not been seen by the cluster for the minimum time specified by `cache_delay`.

Items that are smaller than 16kB are not handled by ICC for performance reasons. The minimum size can only be changed using the `icc_min_size` configuration option in the CLI.

Use the command `set icc_min_size = 16384` to set the minimum object size to be 16384 bytes (= 16kB).

Effect of Node Leaving an ICC Cluster

When a SwiftCache leaves an ICC-enabled cluster it will invalidate some of the cached content. This is because the content stored on the cache will no longer be available to the ICC-enabled cluster.

The proportion of the objects lost from the cache when a node leaves the cluster will be `( 1/n )`, where `n` is the number of nodes in the cluster.

Caches Must Retrieve an Item Once

In an ICC-enabled cluster, each cache must retrieve the item from the origin server once. This then allows each cache to determine whether the item may be handled by ICC (see **Items Not Handled by ICC** above).

For subsequent requests for the same item, each cache will either retrieve the item from the node responsible for storing it or, if the cache *is* the node responsible, simply to return it to the client.

14 Policies

Policies are sets of rules that change the default behaviour of the SwiftCache on a per-request basis.

14.1 Overview

SwiftCache policies have two main functions:

- to optimise the performance of the appliance by enabling it to cache more effectively, or to cache in situations where it might otherwise not;
- to apply configured filters to constrain or modify the end-user browsing experience.

See **Appendix C** for a full list of all of the settings that can be controlled via a policy.

14.2 Introducing Policies

The HTTP protocol provides ways to manage the caching of content. However, it is possible to greatly improve upon these caching methods. SwiftCache policies allow caching behaviour to be customised on a case-by-case basis in order to maximise the potential for saving bandwidth.

Policies allow almost any configuration setting on the SwiftCache to be overridden for requests matching specific rules. The rules are flexible, fully-configurable and can be defined using multiple criteria.

Policies are particularly helpful in the common scenario where websites use semi-dynamic or fully-dynamic URL schemas, which cannot be cached using simple rules alone.

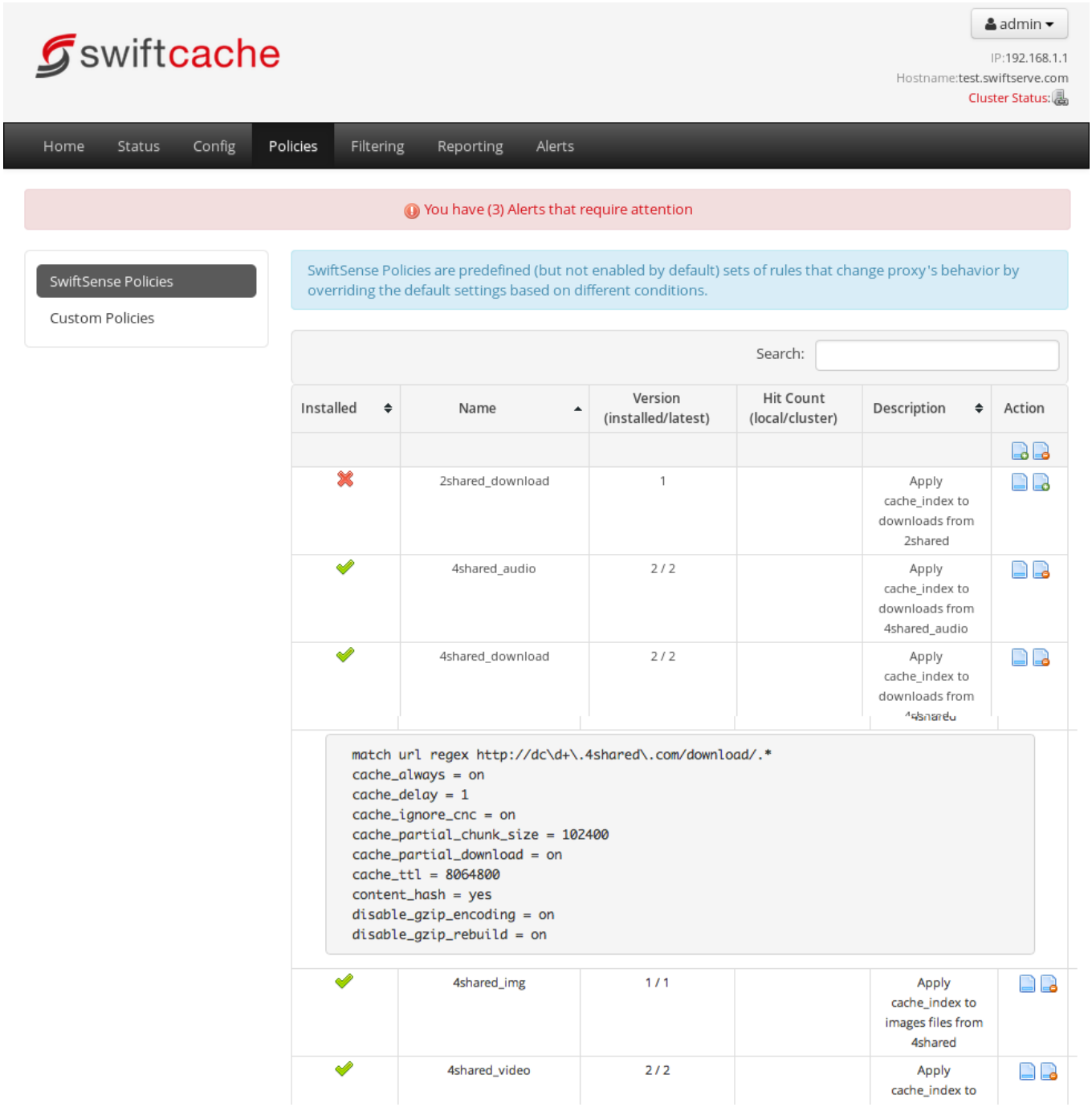
14.3 SwiftSense Policies

SwiftSense is a cloud service that the SwiftCache periodically consults for updated policy recommendations based on its analysis of live traffic. For more detail, please refer to the **SwiftSense** chapter of this manual.

New and updated policies downloaded from SwiftSense are displayed in the GUI separately from local (*Custom*) policies, and are not enabled by default. It is also possible to identify SwiftSense policies in the CLI since their name is prefixed with `upstream_policy-`.

14.3.1 Management

When SwiftSense is enabled it will periodically connect to the cloud service to identify and download new policies. SwiftServe will generate alerts whenever it downloads new policies to notify the operator to review and apply the updates. Operators can view the updates in the *SwiftSense Policies* subsection of the *Policies* tab:



In the example above, the *fileserve* policy has been expanded to allow an operator to review its details and determine whether to install it. It is possible to install individual policies or all policies.

The following columns are shown in this view:

Installed	An coloured icon indicates the current status of the policy: • *Red*: Policy is not installed (disabled). • *Green*: Policy is installed (enabled). • *Blue*: Policy has been edited locally; the local policy takes precedence.
Name	Unique name for this policy
Version (installed/latest)	The version numbers of the policy currently installed and latest version of the policy downloaded from SwiftSense. Updates are not applied to enabled policies until the operator chooses to.
Hit Count	The number of times a policy has been utilised.
Description	Textual description of the policy.
Action	Available options that the operator may take against that policy: • *Show/Hide*: Toggle the display of the policy definition. • *Edit*: Customise installed policy by creating a local copy. • *Install/Uninstall*: Enable or disable the policy.

14.3.2 Recommendations

It is always recommended that SwiftSense policy updates are applied immediately to ensure that optimal caching performance is maintained. Policy updates are tested rigorously against live traffic before being released.

The default behaviour of SwiftSense policies will be suitable for normal operation. However, there are situations when an operator may choose to alter the policy behaviour.

This would be if the operator wanted to extend the expiry time on objects stored for a specific site. While this customisation could increase the cache efficiency, it also increases the chance of serving stale content.

Where operators choose to override the behaviour for a specific site, it is always recommended that these local policies are based on the most recent master policy if one exists. This will ensure that the local policy stays up to date with any improvements that are made via the master policy.

14.4 Custom Policies

Custom policies are defined to optimise caching of traffic in specific scenarios not covered by SwiftSense policies. They are also needed to apply *filter policies* (or *filter sets*).

14.4.1 Filter Policies

It is necessary to create a custom policy to apply a filter policy. The minimum configuration is a *match rule* (see below). Then one or more filter policies can be selected by changing the *Filter Set Names* parameter. This is achieved by clicking on the words, *n filters configured*, and selecting the desired filters. This collection of filter policies is known as the *filter set*.

See the **Filtering** chapter of this document for more information.

14.4.2 Ordering of Policies

SwiftCache evaluates all policies against every request. When a policy matches it applies the settings from that policy. Where multiple policies match a request, each policy is applied in the order configured. If the same configuration key has different values defined in those policies, the value in the final matching policy will be applied.

The order of policies is important to ensure that final behaviour is as expected. Policies can be reordered in the GUI by dragging their names. From the CLI, operators can use the commands:

- `policy <policy_a> move after <policy_b>` and
- `policy <policy_a> move before <policy_b>`

where `<policy_a>` and `<policy_b>` are names of policies.

14.4.3 Match Rules

Every policy must have at least one match rule. The match rule is used to determine the requests for which the policy should be applied. Where multiple matches are specified within a policy, they are applied with the logical `AND` operator. This restricts the number of matching requests as all match rules must be met to trigger the policy.

A match rule is defined as:

```
match <parameter> [negate] <operator> <value>
```

`<parameter>` specifies the aspect of the client request to test for a match. The possible parameters are described in the table below:

`method`	The HTTP request method. Possible values are `GET`, `PUT`, `POST`, `DELETE`, `HEAD` etc.
`client`	The IPv4 or IPv6 address of the client or the netblock from which the client request originated.
`server`	The IPv4 or IPv6 address of the origin server or the netblock in which the server is located.
`url`	The full URL of the request, e.g. `http://<host>:<port>/<path>?<querystring>`
`mime`	The value of the `Content-type` header in the response. This is only available after origin server headers have been processed.
`status`	The HTTP status code for the server response, e.g. `200 OK`. This is only available after origin server headers have been processed.
`content-length`	The value of the `Content-length` header in the response, i.e. the size of the file being downloaded. This is only available after origin server headers have been processed.

`<operator>` control the type of match that is applied against the `<parameter>`. The possible operators are described below:

`matches regex`	Matches the parameter against a regular expression (regex). The regex should match the whole value and not a substring. For more information on regular expressions, please refer to http://www.wikipedia.org/wiki/Regular_expression.
`equal to`	Checks whether the parameter is identical to the value (case-insensitive).
`matches subnet`	Matches the parameter against a comma-separated list of IPv4/v6 networks/hosts, where networks are defined as CIDR blocks e.g. `10.0.0.0/16`.
`in list`	Matches the parameter against a comma-separated list of values.
`value between`	Checks if the parameter is within a range of values (inclusive).
`URL has all params`	Checks if the URL has all the parameters.

Named Captures

Regex matches can be used to capture parts (or *strings*) of policy parameters, typically the URL parameter. These strings can then be used as substitution variables in the settings within that policy.

Although all strings allowed in policies can be rewritten in this way, this syntax is particularly useful when setting the `cache_index`.

By default the `cache_index` uses the URL to reference the stored object on disk. However with many sites where the URLs are semi-dynamic, we need identify the static component of the URL and use that to key the `cache_index`.

The following example shows how the cache_index is defined for the Filesonic site:

```
[upstream_policy-filesonic]

match url regex http://s\d+\.filesonic\.com/download/(?<file>\w+)

cache_index = filesonic.com/$file
```

The syntax `?<file>` is used to define the named capture. The resulting string is referenced within the scope of that policy as `$file`.

14.4.4 Common Settings

cache_partial_download

Allows caching of partial download content for items stored on disk. This can be useful when clients only request partial ranges instead of complete items. (e.g. download managers, some video clients etc.)

If the `cache_partial_download` option is set it will allow caching of partial requests or responses and interrupted transfers. When a partial response is received it will be written to a single disk file in 1MB *chunks*.

SwiftCache remembers which chunks have been seen before so only when a new chunk is seen will it be written to disk. If a download is stopped part-way through a chunk then none of that data will be written to disk. The size of each chunk can be configured by the `cache_partial_chunk_size` key.

complete_download

Allows a SwiftCache to carry on a download when a client terminates the download part-way through.

complete_download is a feature whereby if a client starts a download and then stops the download before it has completed the SwiftCache can carry on the download session so that the entire item can be cached.

It can be configured with the keys: + *complete_download*, which enables and disables the feature; + *complete_download_min_count*, which specifies how many times we need to see an object before we complete the download and cache it; and + *complete_download_threshold*, which specifies what percentage of the object needs to be downloaded before the download is completed.

cache_async_fetch

Allows a Swiftcache to cache content that would normally not be fully downloaded because of the use of range headers.

If `cache_async_fetch` is enabled it operates on connections that fulfil two criteria:

- the client has sent a range request; and
- the object requested is not in the cache.

SwiftCache will then fulfil the original client's request with the range header intact. SwiftCache will also simultaneously reissue the request *without the range header* to the origin server using the credentials from the original client request. This will allow SwiftCache to download the entire object.

This is useful for objects that are commonly-accessed via a range request and rarely downloaded as a complete object. This behaviour can be enabled or disabled globally, and within specific policies.

cache_async_refresh

Allows a SwiftCache to deliver cached content before checking if the item should have expired from the cache.

Normally SwiftCache will check the validity of an item with the origin server before returning it. With `cache_async_refresh` enabled, SwiftCache will deliver cached content immediately without first checking whether the object has expired. SwiftCache will also simultaneously refresh the content in the background.

This can cause expired content to be served so `cache_async_refresh` should be used with caution.

ignore_range

Allows SwiftCache to download an entire file requested with a range header while honouring the range request for the original client.

With `ignore_range` set, when a user requests a file with a range header set, SwiftCache will download the complete file from the origin server, but will then honour the client request and only return the part the user requested with the range header.

This can be useful for sites that deliver video content. While an end-user may only request part of the video file, `ignore_range` allows the entire object to be cached, potentially improving hit rate.

14.4.5 Video Seek

The *Video Seek* section of the policy settings allows video content to be cached more efficiently when the client does not support the range header.

Some video clients do not support the range header. When fast-forwarding through a video the client will issue a request that will contain either a time or byte offset to refer to the point in the video to be played back. Without Video Seek, SwiftCache would not be able to respond to these kinds of requests as the URL would not be understood.

Video Seek supports a number of popular video sites.

14.4.6 Safe Search

SwiftCache supports the option to enforce the Safe Search modes offered by the leading search providers, Google and Bing. These modes are provided by the search engines to allow adult content to be filtered from search results.

When using Safe Search, SwiftCache will override the settings that are specified in the client request and apply the configured level of filtering.

Safe Search is implemented through the `append_query_params` configuration key. This looks for query string parameters in the request. If the specified query string parameter already exists in the request, it replaces the value. If the parameter is not found, SwiftCache appends the key and value.

The information below describes the settings that should be applied for Google and Bing. Note that these configuration keys should be applied within appropriate policies that match the search request URL pattern.

Google

```
&safe=strict
```

Please refer to http://support.google.com/websearch/bin/answer.py?answer=510 for more details of Google's Safe Search feature.

For example:

```
[policy-google_safesearch]

match url regex http://www\.google\.[^/]+/search\?.\*

append_query_params = safe=strict
```

Bing

```
&adlt=strict
```

Please refer to http://www.bing.com/community/site_blogs/b/search/archive/2009/06/04/smart-motion-preview-and-safesearch.aspx for more details of Microsoft Bing's SafeSearch feature.

An example for Bing is listed below:

```
[policy-bing_safesearch]

match url regex [http://www\.bing\.com/search\?q.\*](http://www\.bing\.com/search\?q.*)

append_query_params = adlt=strict
```

15 Filtering

15.1 Overview

SwiftCache provides a number of mechanisms for filtering:

- global black and white Lists;
- Brightcloud (Dynamic URL Classification); and
- filter policy black and white Lists.

These can be used individually or in combination. Filters are applied in a strict order. As soon as a filter is matched the rest of the filtering chain is ignored. The precedence of filters is as follows:

Global Lists > Filter Policy Lists > Filter Policy Dynamic URL Classification

A decision of `Undecided` is returned when the filter does not match the request. Processing then continues to the next filter in the chain.

Filtering provides the operator with the ability to restrict, block or otherwise modify access to a particular site or sites. Most filtering is applied through the use of policies. This provides the operator with fine-grained control over which filtering rules are applied to which of their subscribers.

15.2 White Lists

A *white list* defines a set of URLs that should never be filtered. Typically this would be used for the operator's own site and any content partners that the operator wants to ensure are never blocked.

Default Deny mode will block access to *any* URL that is not included on a white list. This is a very restrictive mode of operation that may be suitable for certain enterprise environments, however it is not typically recommended for normal usage.

Please see later in this chapter under **Filter Policies** for more information on configuring *Default Deny* mode.

15.3 Black Lists

A *black list* defines a set of URLs that should be blocked by SwiftCache. If the root of a URL is specified on the black list, it will also match more specific URLs.

For example, adding `example.com` to the black list would filter any URL requested that matches the regular expression `.*example.com.*`.

When a client request matches a black list filter, the operator has the option to specify a URL to redirect the request to. This may be useful to explain to the end-user why the page was blocked and who to contact for assistance. Alternatively the request can simply be blocked with a server close on the connection.

15.4 Global Black and White Lists

Global black and white lists allow an operator to specify which sites should always be blocked or allowed. The sites are specified using a simple list of hostnames, against which each request is compared for a match.

Global lists are configured in the GUI using the *Global Lists* page of the *Filtering* subsection of the *Filtering* tab.

The *Location* can be specified as an HTTP URL or as a local path to a file on the appliance. The *Refresh Period*

defines how frequently it is updated and the *Redirect Location* indicates where denied requests are forwarded to if required.

15.4.1 Global White List

The *Global White List* specifies URLs that are always permitted, irrespective of the settings in any more specific filter policy. Since it is applied first in the filtering logic any URL that matches this list is always allowed.

15.4.2 Global Black List

The *Global Black List* specifies URLs that are always blocked, irrespective of the settings in any more specific filter policy, but with the exception of a URL that is on the *Global White List*. As above, filtered requests may be redirected to another page or blocked.

The *Global Black List* can be useful when a site has to be blocked by a provider for legal reasons.

15.5 Filter Policies

With the exception of the global black and white lists, all filtering rules are applied through policies. Each instance of a filtering configuration is known as a *filter policy*.

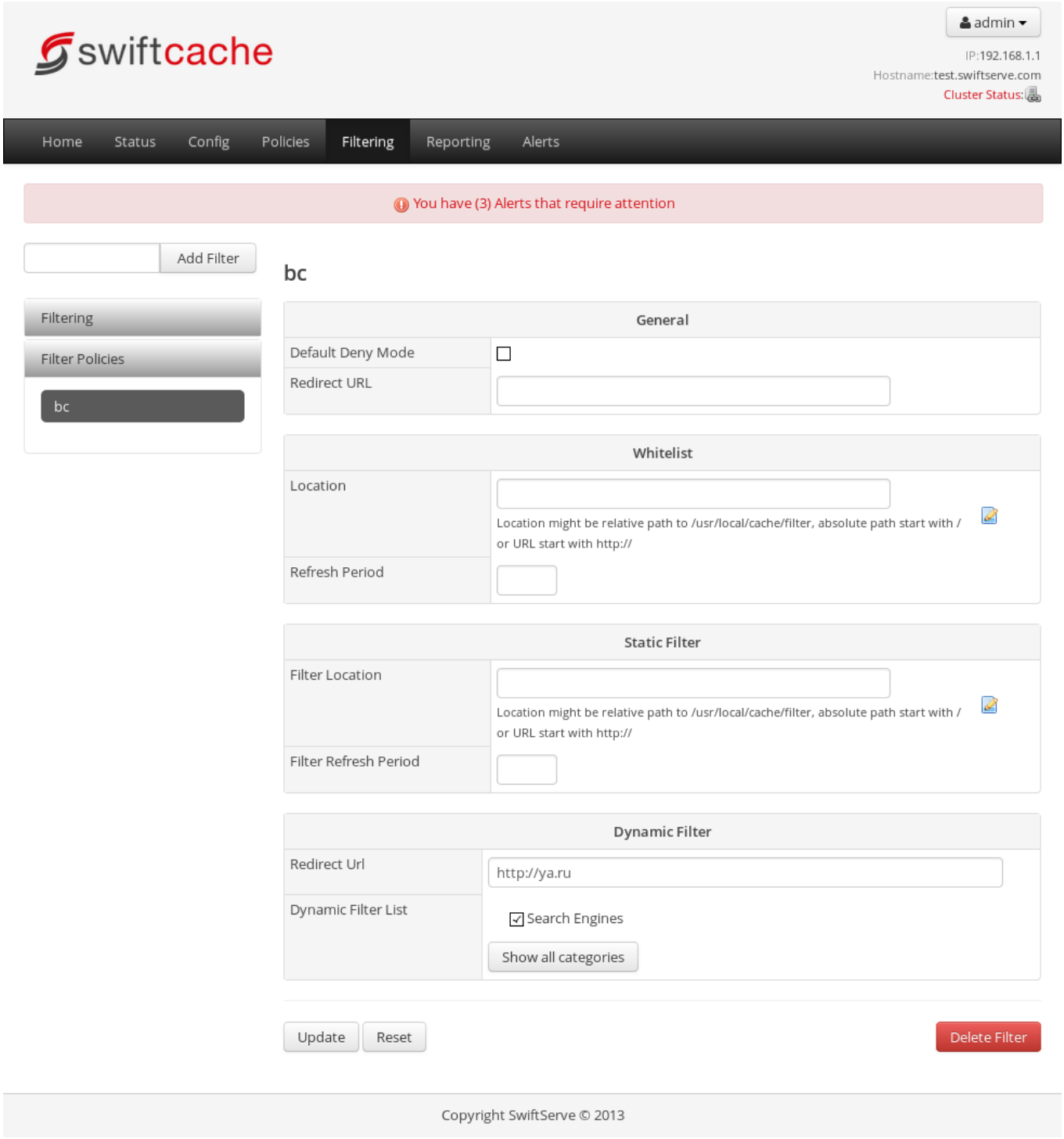
A policy may optionally apply one or more filter policies (the combination is known as a *filter set*). It is possible therefore to define different filtering rules for different groups of end-users using appropriate policies.

Filter policies are managed from the *Filter Policies* subsection of the *Filtering* tab in the GUI.

15.5.1 Terminology

- A filter policy white list is known as a *bypass list*.
- A filter policy black list is known as a *static filter*.

15.5.2 Configuration



The GUI allows operators to create new filter policies and edit the parameters of an existing filter policy.

Existing filter policies are listed on the left-hand side. To add a new filter policy enter the name in the box above and click *Add Filter*.

Each filter policy must be uniquely identified by a name. When a filter policy is applied through a policy it is referenced by this name.

- To view or edit a filter policy click on its name.
- To remove a filter policy click on its name and then use the red *Delete Filter* button at the bottom-right.

The following parameters are always available:

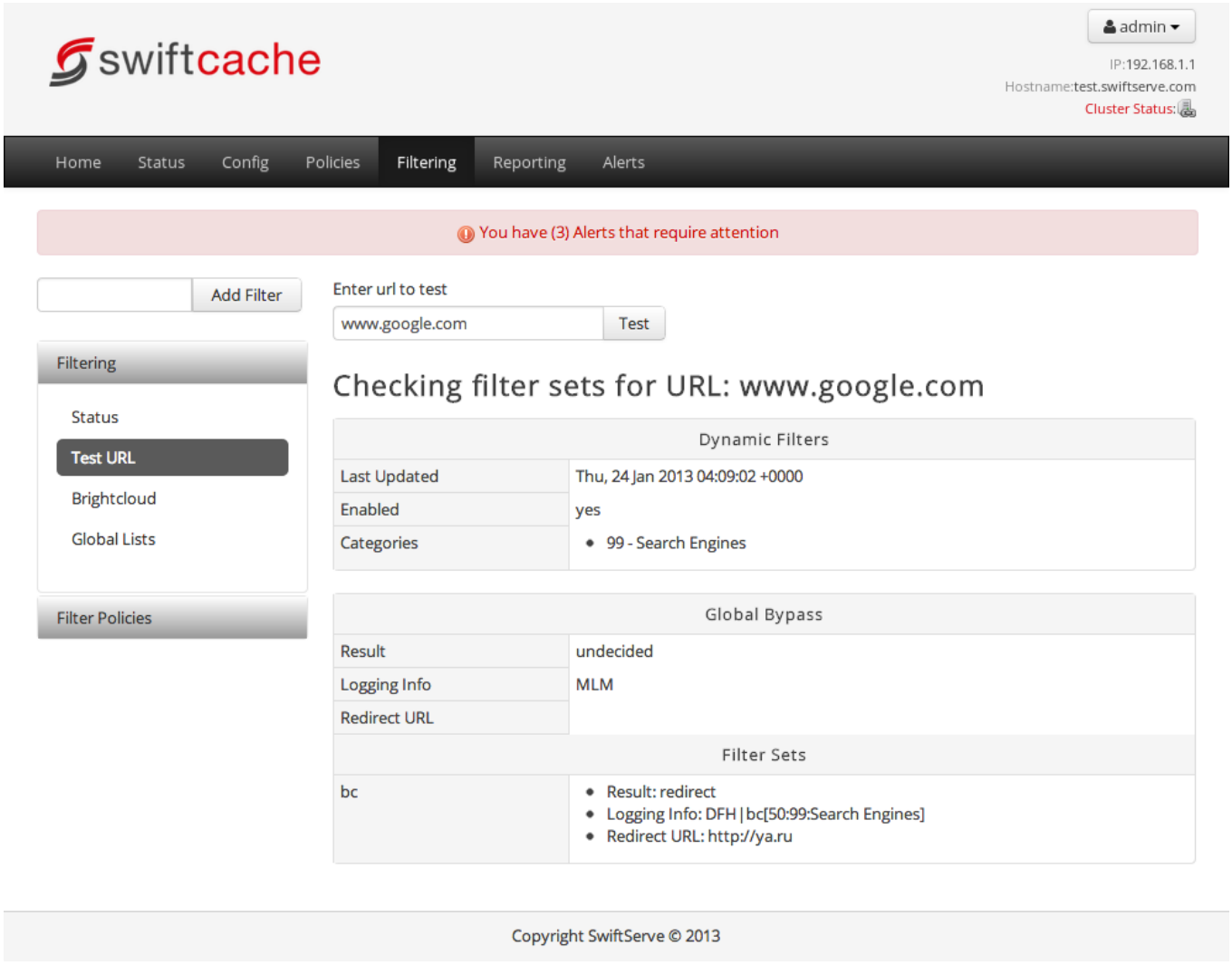
Default Deny Mode	Forces SwiftCache to enter a restrictive mode of operation where all requests are blocked or redirected unless explicitly allowed through inclusion on the global or filter policy white list.
Redirect URL	If specified, requests that match one of the static filters (or are not on the global or filter policy white list in the case that *Default Deny Mode* is active) are redirected to this URL, otherwise they are blocked.
Bypass List Location	HTTP URL or a local path from which the SwiftCache should retrieve the list of URLs to permit within this filter policy.
Bypass Refresh Period	Frequency at which SwiftCache should reload the white list. If set to `0`, SwiftCache will only load the list on startup.
Static Filter Location	HTTP URL or a local path from which the SwiftCache should retrieve the list of URLs to block within this filter policy.
Static Refresh Period	Frequency at which SwiftCache should reload the black list. If set to `0`, SwiftCache will only load the list on startup.

The following parameters are available with the Brightcloud module:

Dynamic Redirect URL	If specified, requests that match one or more of the dynamic filter lists are redirected to this URL, otherwise they are blocked.
Dynamic Filter List	List of categories that should be blocked or redirected. Within the GUI the operator may select from a list.

15.5.3 Filter Status and Testing

An overview of the status of all filters can be viewed on the *Status* page of the *Filtering* subsection of the *Filtering* tab in the GUI provides. The page shows the current total filter statistics in terms of the requests allowed and blocked, and underneath some statistics for the individual filter policies.



An operator can enter a URL and validate it against the full filtering chain in order to determine what action would be taken. This is performed on the *Test URL* page of the *Filtering* subsection of the *Filtering* tab in the GUI. Information is returned about all types of filtering.

15.6 Brightcloud (Dynamic URL Classification)

Dynamic URL classification is a real-time service used to identify the type of content that is being requested. This allows the operator to restrict access based on the nature of the content requested rather than by manually maintaining a list of sites that should be blocked.

This is an optional, add-on feature to SwiftCache and is available via subscription. Access to the service is controlled through the SwiftCache license.

SwiftCache has partnered with Brightcloud (http://www.brightcloud.com) to gain access to the most authoritative and comprehensive source for URL classification. URLs are classified into a series of categories that an operator may choose to block individually.

Dynamic URL classification is typically used by operators to block access to known malware or spyware sites to protect end-users from online threats.

It can also be offered as a personalisation service to consumers, allowing them to define what content they want to block on their own connections. This type of personalisation service requires the SwiftPolicy Manager to control the filtering policies on the SwiftCache. Please contact your vendor for more information.

15.6.1 Categories

The most recent list of categories and their descriptions is available from http://www.brightcloud.com/support/catdescription.php.

15.6.2 Status and Information

Brightcloud will be enabled automatically if the SwiftCache license includes it. An operator may verify that it has been activated by viewing the *Brightcloud* page of the *Filtering* subsection of the *Filtering* tab in the GUI. It is also possible to check the classification category associated with a particular domain on this page.

15.6.3 Usage

Once the Brightcloud filtering module is enabled, it is possible to include dynamic URL classification categories within policies. Please see the **Policies** chapter of this manual for more information.

16 Advanced Features

16.1 Overview

This chapter describes advanced features and configuration settings, and scenarios in which they should be used.

16.2 DNS Resolver

SwiftCache has its own high-performance caching DNS client which can be utilised to reduce the number of connections to DNS servers and improve performance.

When a client connects to a SwiftCache, it is also important for the Swiftcache to carry out its own DNS lookup of the hostname and IP address for the connection. If SwiftCache did not verify the hostname and IP address, then a malicious client could “poison” the HTTP cache by making a false request for a HTTP object from a site under the control of the attacker.

For this reason it is very important and highly recommended that the `always_do_dns` setting is enabled.

The configuration keys related to the SwiftCache DNS client are:

`always_do_dns`	Always do a DNS lookup even if it could be avoided by trusting the client.
`dns_servers`	The list of DNS nameservers to use.
`dnscache_enabled`	Enables the local DNS cache.
`dnscache_size`	The maximum number of entries to be kept in the local DNS cache.
`dnscache_resolve_timeout`	The number of seconds after which a DNS resolve is considered to have failed.

For more information about configuration keys, please refer to **Appendix C**.

16.3 Asymmetric Routing

It is important that any traffic sent from the SwiftCache appliance is routed back via the cache. If not, the resulting asymmetric routing will disrupt client connections. Typically this can occur when SwiftCache is deployed into an ISP network where a local CDN may be installed.

There may be multiple internal routes back from the local CDN servers where the return path will bypass the cache. In this scenario it is important that asymmetric mode is configured and enabled.

For more information on avoiding asymmetric routing, please refer to the **Deployment Scenarios** chapter in this manual.

16.3.1 Enabling Asymmetric Mode

This is done by enabling the configuration key `asymmetric_mode`. SwiftCache will also require the `divert_helper` module to be installed for this functionality to work.

16.4 Fast Disks

Fast Disks is an optional, add-on feature to SwiftCache that needs to be enabled in the license key. Please contact your vendor if you require this feature.

16.4.1 Improving Disk Seek Time

Fast Disks allows an operator to use solid-state disk (SSD) storage with a fast seek time to improve cache performance. A traditional server-grade spinning hard disk (i.e. has moving parts) has a high seek latency (8 milliseconds) in comparison with SSD storage (0.1ms) or an in-memory cache (10 nanoseconds). Disk write operations will also be slower on hard disks with moving parts.

This means that retrieving an object from *memory* is in the region of:

- 800,000 times faster than retrieving it from hard disk; and
- 10,000 times faster than from SSD.

This also means that a *SSD* has a seek time that is approximately 80,000 times faster than that of a hard disk. SSD storage and in-memory caches present a significant performance improvement.

16.4.2 Enabling Fast Disks

A SwiftCache license key enabling use of Fast Disks is required. To use a disk as a fast disk, enable the `fast_disks` configuration key. The `fast_disks` key uses the same configuration syntax as the `cache_disks` option, taking a list of fast disk partitions as its value.

An operator can define the minimum size of a fast disk object with the configuration key `cache_fast_disk_object_size`.

16.5 Trust X-Forwarded-For HTTP Header

The `trust_x_forwarded` configuration key is used to strip or keep `X-Forwarded-For` HTTP headers when used with the `enable_x_forwarded` setting. It may be required depending on the type of Layer 4-7 switch or load balancer being used.

Please refer to the **Deployment Scenarios** chapter in this manual for more information on load balancers.

16.5.1 Enabling Trust X-Forwarded For

If `trust_x_forwarded` is disabled then the client's untrusted `X-Forwarded-For` HTTP header is removed. If enabled, then the trusted X-Forwarded-For is preserved.

If the `enable_x_forwarded` configuration key is enabled then the client's IP address is appended to the `X-Forwarded-For` HTTP header if already present. If the `X-Forwarded-For` header is not present, it will be added using the client's IP address as the value.

16.6 Return to Sender

In some deployments a single SwiftCache may have two or more physical network interfaces connected to a single Layer 4-7 switch or load balancer. In this scenario, each network interface will have its own distinct IP address.

To the Layer 4-7 switch or load balancer, it will appear that it is delivering connections to two separate web caches. It is therefore important that SwiftCache replies to the load balancer via the same physical network

interface as the original request. If not, the load balancer will drop return packets because the reply has returned via wrong interface.

16.6.1 Enabling Return to Sender

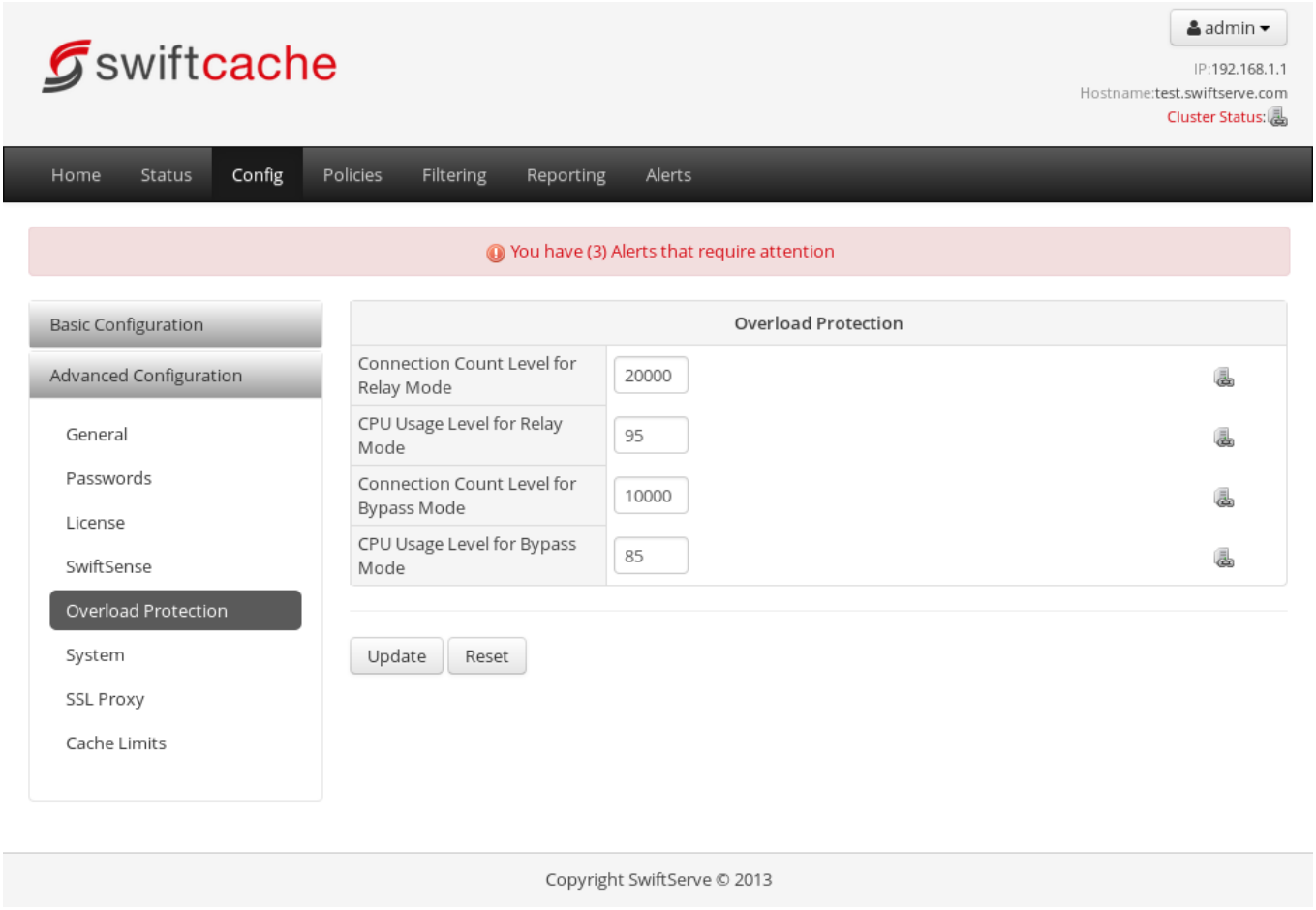
The `return_to_sender` configuration key ensures that associated requests and replies use the same network interface. This has the added benefit of also supporting the scenario of a single SwiftCache being deployed with each physical network interface connected to a different load balancer.

16.7 IPv6 Support

SwiftCache supports deployment on an IPv6 network. All network configuration allows IPv4 and IPv6 address specification. The SwiftCache DNS client supports `A` and `AAAA` records.

In explicit mode, SwiftCache can act as an IPv6 to IPv4 gateway. However in transparent mode, where SwiftCache is spoofing the client's IP address, SwiftCache cannot provide this IPv6 to IPv4 gateway function.

16.8 Overload Protection



There are two levels of overload protection in SwiftCache: *Bypass* and *Relay* modes. These are configured in the GUI on the *Overload Protection* part of the *Advanced Configuration* subsection on the *Config* tab.

When the configured thresholds are reached then the overload protection modes will be activated until the load decreases again.

16.8.1 Bypass Mode

Bypass Mode limits the most resource-intensive aspect of the cache engine: disk access.

Activation of Bypass Mode is determined by CPU load or simultaneous connections thresholds. After either of these thresholds is breached, new requests will be handled in Bypass Mode. Existing requests will be unaffected.

Whilst a connection is in Bypass Mode the cache will not attempt to read or write objects to or from disk, irrespective of any other policy settings. This behaviour is very similar to the `cache_never` configuration key, except that it is applied dynamically when under exceptionally high load.

However, SwiftCache will continue to apply policies, filters and other connection logic as per normal in Bypass Mode. This ensures that configured business rules continue to be enforced even during exceptionally high load.

16.8.2 Relay Mode

Relay Mode is applied as a last resort when the load on the cache is so severe that the amount of processing needs to be drastically reduced.

Relay Mode is complementary to Bypass Mode, and is similarly activated by CPU or simultaneous connection thresholds.

Whilst in Relay Mode, SwiftCache acts as a traffic router. It not only avoids any disk access, but also most connection processing. This means that policies, filtering and other connection logic is not applied until the load decreases again.

16.9 SSL Support

Swiftcache has several different levels of support for SSL. For SSL traffic there are two options available:

SSL proxy mode	This enables caching of objects that are requested and served via SSL connections. This is an optional module and needs to be enabled in the license key to operate.
SSL relay mode	SwiftCache routes SSL connections between the client and server, but no caching is possible as SwiftCache cannot inspect the encrypted data.

Swiftcache also offers SSL as an option for securing the web administration interface (GUI). Please refer to the **Securing Swiftcache** section in the **Operations Guide** chapter.

16.9.1 SSL Proxy Mode

SSL Proxy Mode is an optional, add-on feature to SwiftCache that needs to be enabled in the license key. Please contact your vendor if you require this feature.

SSL Proxy Mode gives organisations the option to cache encrypted web traffic as well, resulting in additional bandwidth savings.

An organisation may have a policy to route all web traffic through a web cache to save on bandwidth usage and provide traceability for clients. Ordinarily only unencrypted (non-SSL) HTTP traffic can be cached. This is because HTTPS traffic is encrypted to prevent payload inspection during transit between the client and the origin server.

Distribution of SSL Certificates

To use this mode, it must be possible for the organisation to distribute SSL certificates to the clients though existing IT management processes, for example via Group Policies within a Microsoft Windows Domain

environment.

The ability to supply SSL certificates to clients is necessary because to operate SwiftCache as a SSL cache, clients must trust the SSL server certificates generated dynamically by SwiftCache. This in turn means that the SSL Certificate Authority (CA) certificate used to sign the server certificates must be installed onto any client that will be routed via SwiftCache.

Re-establishing a Chain of Trust

When in SSL Proxy Mode, SwiftCache pretends to be the origin server to the client by issuing itself a SSL server certificate for the origin server's domain. It signs the server certificate using the CA certificate.

Ordinarily this would not be trusted by the client. However, because the client has also installed and trusts the same CA certificate, the chain of trust is re-established.

Valid Certificate Workflow

When a client requests content over HTTPS, Swiftcache creates a new connection to the requested site and checks the SSL certificate provided by the origin server. If the SSL certificate from the origin server is valid then SwiftCache will generate a “good” SSL certificate for the origin server's domain. SwiftCache then uses this to encrypt its own connection to the client.

The client trusts the CA that SwiftServe has used to create the certificate and the certificate matches the domain name the client originally requested, so the client does not generate any web browser warnings.

Swiftcache then will pass any requests from the client to the real SSL origin server using its encrypted connection to the origin server. SwiftCache delivers any content from the origin server back to the client via its separate encrypted connection to the client.

Invalid Certificate Workflow

If the origin server's SSL certificate is invalid for any reason (e.g. it is self-signed, has expired etc.) then SwiftCache will deliberately generate a “bad” certificate to encrypt the connection back to the client. This will cause the client web browser to generate a SSL certificate warning in much the same way as if the client had connected directly to the origin server.

If the end-user observes the warning, the connection will be terminated. If not, then Swiftcache will pass back encrypted content to the client as described above.

Note that if an end-user carefully examines the SSL certificates generated by SwiftCache, then the end-user could become aware that the certificate has been signed by the SwiftCache Certificate Authority rather than the origin server.

Enabling SSL Proxy Mode

A SwiftCache license key enabling use of SSL Proxy Mode is required. SSL Proxy Mode is enabled in the GUI in the *SSL* part of the *Advanced Configuration* subsection of the *Config* tab. This will only be visible if the feature is enabled in the license key. To display the “good” SSL CA cert and key you can click the notepad icon on this screen.

SSL Proxy Mode is enabled in the CLI with the command `set enable_ssl_proxy on`.

Other options for SSL connections are configured via the CLI. To alter the time-out for SSL connections, operators can change the default value of thirty seconds in `timeout_ssl_handshake`.

To control how many SSL certificates are stored in memory, change the value of the `ssl_cache_cert_size` configuration key.

To store SSL certificates on disk then operators will need to set a directory path using the `ssl_cache_cert_path` configuration key.

SSL Proxy Mode and Overload Protection

If a SwiftCache enters Bypass or Relay Mode (see **Overload Protection** above) then this will disrupt normal SSL proxy operation. Any client connections over SSL will receive the SSL certificate of the origin server until SwiftCache leaves Bypass and Relay Mode.

If the client already has an active SSL connection, the web browser may detect the change in certificate as a potential security problem and may deliver a warning message to the end-user.

16.9.2 SSL Relay Mode

In more usual operation (e.g. an ISP environment), a SwiftCache should not receive any SSL connections as it does not listen on port 443 (the default port for HTTPS traffic).

If a SwiftCache encounters HTTPS traffic on port 80 then it will handle that connection via relay mode and will proxy the connection. However, as SwiftCache cannot inspect the encrypted payload, it will not cache the traffic.

Note that relaying SSL will consume CPU and network resources available to SwiftCache, and will have no corresponding benefit through caching of content.

16.10 Limiting Download Rates

It is possible to limit the download rate of client connections within SwiftCache. However, this feature does not allow limiting of the number of connections opened by a client.

An operator can limit the download rate of either an individual HTTP connection from a client or apply the download rate limit as a total maximum allowance per client IP address.

Limits can be applied globally to all clients, or under specific conditions using a policy. This can be useful when an operator wants to limit the download rate for specific clients, for specific sites or by server.

There is an option availablle for *burst mode* so that an operator can allow a specified amount of data to be transferred at full speed before the rate limiting begins. Note that burst mode is only applied to each connection and is not applied per IP address. This means that multiple short requests will not be limited.

This option is useful if an operator does not want to affect normal web browsing, but wants to limit large downloads from consuming a large proportion of available bandwidth.

16.10.1 Enabling Rate Limiting

Download rate limits are configured via the CLI. Some configuration examples are provided below:

`set ratelimit_type ip`	Apply rate limit per client IP address (default setting).
`set ratelimit_type connection`	Apply rate limit per client connection.
`set ratelimit 300`	Limits the download rate for each client to 300 kilobits per second (kbps).
`set ratelimit_burst 2000`	Starts to limit the rate of downloads after 2000 kilobytes (kB) of data have been transferred.

Note that if limiting by connection then this may permit end-users to use download managers to exceed their quota.

17 SwiftSense

17.1 Overview

SwiftSense is a cloud service for SwiftCache that analyses live traffic to optimise performance and provide business reporting tools. It also provides a way to fine-tune caching policies across the entire network to ensure that the maximum caching efficiency is being achieved.

When enabled, SwiftSense collects anonymised summary statistics from each SwiftCache to identify which sites are popular. The service then suggests new policies that will improve the caching behaviour of SwiftCache. SwiftSense uses intelligent analysis to ensure that the policy it creates is relevant to that individual cache rather than wasting resources on processing policies that will not generate bandwidth savings.

Updated policies are downloaded by each SwiftCache periodically and are available for review by the operator. The new rules are not applied automatically to ensure that the operator retains full control over the configuration and operation of their SwiftCache. For example an operator may choose to override behaviour on a subset of sites or choose not to apply the SwiftSense recommended policies.

17.2 Security

All data stored on SwiftSense and communications with SwiftSense are secured to prevent unauthorised access to the data. An SSL-encrypted transport layer is used to secure all communications with SwiftSense.

Data held by SwiftSense is stored in secure data centres and access to the data is controlled through a secure web interface.

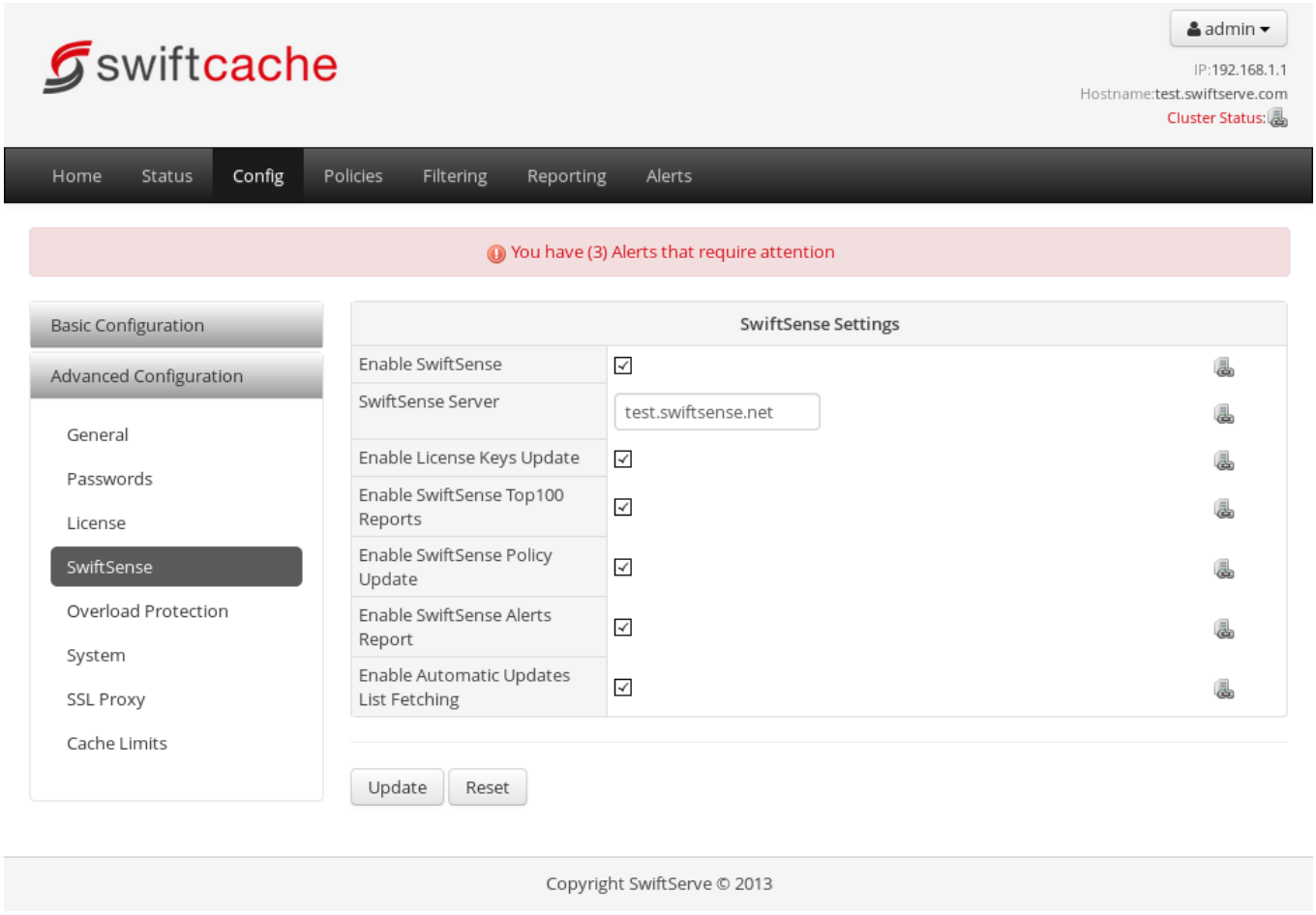
17.3 Functions

SwiftSense provides four functions. It is recommended that all four functions are enabled to ensure optimal performance, however if required an operator may choose to disable one or more of these functions:

License Key Update	When license update is enabled, SwiftCache will periodically contact SwiftSense to check if a new license key is available. The license key update feature allows for the auto-provisioning of license keys, for example to enable a license controlled piece of functionality.
Top100 Reports	The top 100 reports feature enables the feedback loop of SwiftSense in order to ensure that the policies are optimised for the traffic pattern on the SwiftCache. When enabled the SwiftCache will periodically report anonymous performance and efficiency statistics to SwiftSense.
Policy Update	When policy update is enabled, the SwiftCache will periodically contact SwiftSense and request new policies. If new policies exist they will be downloaded to the SwiftCache.
Alerts Report	When alerts report update is enabled, the SwiftCache will send all new alerts to SwiftSense in order to enable centralised monitoring.

17.4 SwiftSense User Interfaces

17.4.1 SwiftCache GUI



SwiftSense settings are controlled from the *SwiftSense* part of the *Advanced Configuration* subsection of the *Config* tab in the GUI.

17.4.2 SwiftSense Web Interface

The SwiftSense web interface consists of the following tabs across the top: *Dashboard*, *Caches*, *Organisations*, *Reporting*, *System* and *Admin*. These are described in more detail below.

At the top-right there is a link to the *My account* page, where an operator can change the account password and displayed name, and a *Logout* button.

17.4.3 Dashboard

The *Dashboard* tab displays a summary graph for aggregate traffic along with byte and request hit rates for all deployed caches, with a choice of day, week, month or year views.

Underneath is a table listing the most recent statistics for the caches by organisation:

- *Alive caches*
- *Total caches*
- *Users*
- *Traffic*
- *Requests per second*

- *Hit rate (bytes)*

17.4.4 Caches

The *Caches* tab displays a table of information about all caches:

- *Name*
- *IP Address*
- *Organisation*
- *Traffic*
- *Requests per second*
- *Hit rate (bytes)*
- *License State*
- *Version*

It is possible to group caches together. To do this, select multiple caches to create a group or click on the *Groups* page to view and configure existing groups.

Clicking on an individual cache will allow operators to view and edit its profile. It is also possible to set the organisation of one or more caches from the main table.

17.4.5 Organisations

The *Organisations* tab displays a table of the organisations. It also allows operators to add a new organisation and to edit or delete an existing organisation.

Clicking on the *Users* page will display a table of the users. It also allows operators to add a new user and to edit or delete an existing user.

17.4.6 Reporting

The *Reporting* tab shows aggregated data for all caches (or a chosen group), accross the following sub pages:

- *Top sites*
- *Top categories*
- *Top videos*
- *Cache stats*
- *Cache alerts*
- *Cache count*
- *Traffic sources*
- *Cache activity*
- *Cache geography*
- *Policy stats*
- *Traffic*

17.4.7 System

The *System* tab has two pages:

Action log	Shows an audit trail of all user actions.
Policies	Allows an operator to add new policies and to edit or delete existing ones.

17.4.8 Admin

The *Admin* tab has two pages:

Users	Shows a table of the user accounts. It allows an operator to add new users and to edit or delete existing ones.
Cache RPM files	Shows the current list of cache software packages. It allows an operator to upload a new package and to delete existing ones.

18 Troubleshooting

18.1 Diagnostics

The following actions are recommended for diagnosing problems:

- Look for unexplained dips on cache graphs.
- If a cache graph is a flat line, look at available upstream bandwidth and network switch statistics.
- Check that the all expected processes are running.
- Check for new alerts.

18.2 Support

To contact the SwiftCache support team and open a ticket, please send an email to support@swiftserve.com.

19 Appendix A: Log File Format

Log File Format

Last updated up to version 2.4.6 (rc)

We support a customizable log file format. The format can be configured with the config option "access_log_format". The default value (if missing) is "%n %x %T %d %t %a %c %m %i %u %l %f".

Each of the letters prefixed by '%' will be replaced with the relevant info as explained below. Everything else in the log format string will be ignored and output "as is".

Placeholder	Description	Available since
%a	http status code (numeric)	2.0
%b	user agent	2.0
%c	length [total bytes sent to client]	2.0
%C	C32 sum of response (not always available)	2.4.7
%d	request duration (seconds)	2.0
%D	date in proxy own format (as in error.log)	2.4.7
%e	MIME Type	2.0
%f	filtering info [see below]	2.0
%h	request completion date/time human readable format (local timezone)	2.0
%i	client ip	2.0
%l	log info [contains various string separated by '-' see below]	2.0
%m	http method	2.0
%n	request completion date/time in timestamp format (the number of seconds since the epoch)	2.0
%o	original endpoint (IP address client was connecting to)	2.1.2
%p	How much bytes were served from the origin server in case of PARTIAL_HIT status.	2.2
%P	Matched policies list (comma-separated)	2.4.1
%r	Requested range	2.4.7
%R	cache reader index	2.1.7
%s	server ip [it can be another cache peer in case of ICC hit]	2.0
%t	log status [see below]	2.0
%T	TTFB (time to first byte) - the duration (seconds) since the last byte of the client request was processed until the first byte of response was sent. This value is meaningless if relay mode is used for an invalid request.	2.3.1
%u	request url	2.0
%W	cache writer index	2.1.7

%x	debug connection id - 8 hexadecimal between square brackets - same one used in the debug log	2.0

Log Status

Status	Description	Available since
OP_CRELAY	overload protection - relay - cpu level	Before 2.3
OP_SRELAY	overload protection - relay - session count	Before 2.3
RELAY	relay mode	Before 2.3
TCP_CBP_MISS	overload protection - cache bypass - cpu level	Before 2.3
TCP_CNC_MISS	client headers prevented caching	Before 2.3
TCP_HIT	cache hit, served from cache	Before 2.3
TCP_ICC_HIT	peer cache hit, served from a peer cache	Before 2.3
TCP_ICC_MISS	peer cache miss	Before 2.3
TCP_MISS	cache miss	Before 2.3
TCP_PARTIAL_HIT	partial content hit (for incomplete cached objects)	Before 2.3
TCP_PARTIAL_MISS	partial cache file was found, but requested range is not yet cached	2.3.10
TCP_REFRESH_HIT	cached content revalidated and served from cache	Before 2.3
TCP_REFRESH_MISS	our cache was no longer valid, the content has been retrieved again from the server	Before 2.3
TCP_REFRESH_MISS_METADATA	server sent 304 on our cached content but metadata has changed and cache is invalidated	Before 2.3
TCP_RNC_MISS	revalidate aborted due to config rules	Before 2.3
TCP_SBP_MISS	overload protection - cache bypass	Before 2.3
TCP_SNC_MISS	server headers prevented caching	Before 2.3

Log Info

Flag	Description	Available since
ALCON	using an existing server connection	Before 2.3
BLCK	blocked connection due to a policy rule	Before 2.3
BLCKAUTH	blocked by authd	Before 2.3
BLCKEX	blocked explicit connection	As of 2.4.5
BLCKNS	blocked Netscaler connection	As of 2.4.5
BLCKFT	fetch session blocked due to a policy rule	Before 2.3
C32	partial content hashing was used	Before 2.3
CALL	the request hit a cache_always rule	Before 2.3
CD	Complete_download option was specified, client connection was interrupted but download was completed to cache.	Before 2.3
CHNK	the response is chunked encoded	Before 2.3
CI	cache index, cache_index config option	Before 2.3
CL	the response has a Content-Length header	Before 2.3
CLARGE	content was too large to be saved (see max_cache_object_size)	Before 2.3
CLTRNGE	range request - served from cache	Before 2.3
CNVR	the request hit a cache_never rule	Before 2.3
CRCBP	cpu level bypass mode enabled - cache read operation ignored	Before 2.3
CSEEK	video seek request policy used	Before 2.3
CSTERR	custom error response	Before 2.3
CWCBP	cpu level bypass mode enabled - cache write operation ignored	Before 2.3
EXCO	explicit type connection/request (direct connection/absolute url)	Before 2.3
EXRNG	Client's request range was extended before passing to server (cache_partial_extend_range=on)	2.4.9
F4VSEEK	seek request in a f4v - served from cache	Before 2.3
FLVSEEK	seek request in a flv - served from cache	Before 2.3

FNWR	Writer creation disabled because the related background shared server fetch (share_server_connection=on) could not create the writer. E.g. cache_delay not reached etc	2.4.5
ICAP	the request went to an ICAP server for response modification	Before 2.3
INCO	intercepted connection/request	Before 2.3
INVCS32	32KB prefix checksum validation failed for partially cached file, which caused file invalidation	As of 2.4.1
INVTRUNC	corrupted (truncated) cache object encountered and is invalidated (just TRUNC before 2.4.7)	2.4.7
INVMETH	cache item was invalidated because non-GET request to the url was encountered	2.4.7
INVPURGE	Cache object was purged due to config (purge_older_than option)	2.4.7
INVPDD	Sparse cache object was deleted because partial download is disabled	2.4.7
INVCCRERR	Cache item was invalidated because of exception during reader creation	2.4.7
INVRFRMD	Cache item was invalidated because of changed response metadata (refresh metadata miss)	2.4.7
INVRFR	Cache item was invalidated because of refresh miss	2.4.7
NOCS32	Cache item was ignored due to missing C32 checksum	2.4.7
INTR	the connection closed before we could complete the transfer	Before 2.3
KA	the connection with the client was kept-alive	Before 2.3
NCERR	not cached error (504)	Before 2.3
NODNS	the original endpoint is used; no dns lookup is performed	Before 2.3
NFDS	no available file descriptors when connecting to server	2.4.6+
NPRT	no available local ports when connecting to server	2.4.6+
NP	the server doesn't support partial content (used with a range request)	Before 2.3
NSCO	Netscaler type connection/request (direct connection/relative url)	Before 2.3
QSR	queue size exception (for async io, the disk load is too high) - cache object lookup aborted	Before 2.3
QSW	queue size exception (for async io, the disk load is too high) - cache object creation aborted	Before 2.3
RDCT	redirect (302) response	Before 2.3
RELAY	we switched to rely mode	Before 2.3

RENERR	entity error (413)	Before 2.3
RIDXD	a valid cache object was found at the non-gzip index	Before 2.3
RIDXGZ	a valid cache object was found at the gzip index for a client supporting gzip	Before 2.3
RNGERR	range error (416)	Before 2.3
RWRT	url rewritten due to a policy rule	Before 2.3
SNC	the server prohibits caching	Before 2.3
SRCBP	session count bypass mode enabled - cache read operation ignored	Before 2.3
SWCBP	session count bypass mode enabled - cache write operation ignored	Before 2.3
TRSP	transparent (spoofing) mode	Before 2.3
WIDX32	write cache object from do32	Before 2.3
WIDXD	write cache object at the non-gzip (default) index	Before 2.3
WIDXFGZ	write cache object forced at gzip index. The server is sending identity, the client supports gzip. We send it as received and save it gzip at the gzip index.	Before 2.3
WIDXGZ	write cache object at gzip index. The server is sending it gzipped we send/save it as we receive it	Before 2.3
WIDXMEM	the object was saved in memory (non persistent)	Before 2.3
WIDXUNK	an unknown/invalid writer model/state - it should never occur	Before 2.3
BRERR	bad request error (we send a 400 response)	2.3+
DUPCHDR	duplicate client headers entries found with different values that we cannot reliable handle	2.3+
DUPSHDR	duplicate server headers entries found with different values that we cannot reliable handle (CL for instance)	2.3+
FORCEREL	force_relay config option was evaluated to true	2.3+
HSTER	we cannot deduce the host information for an explicit connection	2.3+
LHHR	we have 'Host: localhost' header or similar, we use the original endpoint and ignore the header entry	2.3+
LRGCHDR	large client headers	2.3+
LRGSHDR	large server headers	2.3+
NIMPLERR	not implemented (we send a 501 response)	2.3+

NIP	client sent request to a different IP than the one proxy resolved, (see also IIP)	2.3+
IIP	client sent request to the same IP as proxy resolved (see also NIP)	2.3+
REQT	client timeout on the first byte of the request (usually encountered in 'server speaks first' protocols)	2.3+
RFDSK	a fast disk was used for read (should occur along RIDX* strings)	2.3+
SPARSEERR	corupted sparse object encountered	2.3+
TCP_RNC_MISS	re-applying the config on an existing cache object (using the cached headers) caused the cache_never to get set	2.3+
TRUNC	corrupted (truncated) cache object encountered; renamed to INVTRUNC in 2.4.7	2.3+
WFDSK	a fast disk was used for write (should occur along WIDX* strings)	2.3+
URELINTR	request was blocked as unwanted relay	2.3.9
PURGE	Cache object was purged due to config (purge_older_than option); renamed to INVPURGE in 2.4.7	2.3.10
CNC	Client sent no-cache headers	Before 2.3
IPRATE	Rate limiting by IP is enabled	2.3.10
CRATE	Rate limiting by connection is enabled	2.3.10
FRI0	Filtering bypass enabled due to cookie in the request	2.3.7
FRI1, FRI2, FRI3	Error decoding filtering bypass cookie	2.3.7
SEEKRNGE	Seek request with seek_type=range was served from cache (see #5107)	2.4.1
STALE	cache hit was served from expired object because of cache_async_refresh=on	2.3.11
FETCH	async fetch operation has been started to retrieve and store the full object (or refresh the object in case of cache_async_refresh=on)	2.3.11
RFETCH	async fetch operation has been started to retrieve and store the range extended to have only full partial chunks	2.3.11
ICC	ICC request from other cache from cluster	2.2.6
ICH	ICC peer cache hit	Before 2.3
ICM	ICC peer cache miss	Before 2.3
ICSM	A local cache object found which is bound to another ICC node; initiated a move operation	2.4.1

ICS	Seen a request which is bound to another ICC node; notified the other node	2.4.1
TOCS	Timeout while receiving the start of request (will appear only if we don't try to relay)	2.3.11/2.4.1
TORC	Timeout during a client read operation	2.3.11/2.4.1
TORS	Timeout during a server read operation	2.3.11/2.4.1
TOWC	Timeout during a client write operation	2.3.11/2.4.1
TOWS	Timeout during a server write operation	2.3.11/2.4.1
TOCO	Connect timeout	2.3.11/2.4.1
TOKA	Keep-Alive timeout (we don't have an access log entry for this request)	2.3.11/2.4.1
TOFT	Filter timeout (we proceed with the request processing)	2.3.11/2.4.1
TORE	Relay timeout	2.3.11/2.4.1
TOIO	Disk I/O timeout	2.3.11/2.4.1
TOUN	Unspecified timeout (followed by the numeric value)	2.3.11/2.4.1
RSSLDC	Connection dropped (reverse ssl proxy/invalid origin certificate)	2.4.1
RSSLRD	302 redirect response (reverse ssl proxy/invalid origin certificate)	2.4.1
RSSLGE	Gateway Error response (reverse ssl proxy/invalid origin certificate)	2.4.1
RSSLIG	The invalid origin certificate is ignored (reverse ssl proxy)	2.4.1

Filtering Info

- BCD - bc disabled
- DEF - default decision [allow - end of decision chain]
- DFH:cat_id:score[category name] - dynamic hit - category id score and name included
- DFM:cat_id:score[category name] (1 or more categories) - dynamic miss along the url associated categories and score
- DFN - dynamic category new (we never seen this before, a request for brightcloud category info was initiated)
- DFU - dynamic category unknown (brightcloud can't classify it)
- DFI - dynamic category ignored (brightcloud doesn't classify it and we don't query the live service or store it in the local database). Added in 2.3.9, currently used only for ipv6 addresses.
- FD - filtering disabled
- FTO - filtering module timeout - request processing forced without waiting for filtering anymore

- LSI - local filter set has an invalid name - set checks are skipped
- MWH - meta whitelist hit - allow the request (replaces WFH|META)
- MBH - meta blacklist hit - block/redirect (added in 2.3.7)
- MLM - meta lists miss - continue down the chain (added in 2.3.7 - replaces MWM)
- SFH - static filter hit (followed by filter name)
- SFM - static filter miss
- SPD - spm disabled
- SPSI - spm static invalid - dynamic spm check skiped [replaces ST_INV|SPM]
- SPBI - spm bc invalid [replaced BC_INV|SPM]
- WFH - white filter hit
- WFM - white filter miss
- BP - bypass used, only the global lists were checked
- FR0 - the cookie was invalid, we aborted the cookie redirects
- FR1 - first step cookie redirect
- FR2 - second step cookie redirect

20 Appendix C: Configuration Key Reference

Below is a table listing all the configuration keys.

Name	Type	Description
access_log_format	string	%a - status code %b - user agent (browser information) %c - client content length %C - C32 sum of the response (if available) %d - request duration %t - status (text) %D - timestamp in error.log format %e - mime type %f - filtering info %h - human readable timestamp %i - client ip %l - log info %m - method %n - unix timestamp; %o - original_endpoint %p - partial server in %P - matched policies %r - requested content range %R - cache read index %s - server ip %t - log status %W - cache write index %T - time to first byte %u - url %x - connection tracking id NOTE: %b and non-space delimiters/missing delimiterscan lead to performance drawbacks
admin_password	string	Password for admin user
admin_port	int, r80-65535	Port that web admin interface listens on
admin_secret	password	Secret used for internal communication. Must be the same on all machines in the cluster
admin_ssl	bool	Should admin use SSL/TLS
alert_filter_time_delta	int	Time delta in seconds to not show alerts in GUI that older then current time minus time delta (not showing alerts older then 7 days by default). Set to 0 to turn filtering off.
alert_notification_bar_min_severity	int	Minimal severity to show alerts in notification area
alert_notifications	string	A JSON encoded list of alert notifications, in the format of [{'alert': 'os.io.cpu', 'severity': 'high', 'action_type' : 'mailto', 'data': 'admin@example.com'}, ...]
alerts_retention_period	int, ~+	Maximum age of retained alerts (days). Zero means never delete
allow_connect	bool	Allow clients to use CONNECT method.
allow_explicit	bool	Allow direct connections from browsers that have an explicit proxy setting
allow_loop_connections	bool	Allow connections to proxy box. DANGEROUS.
allow_netscaler	bool	Allow direct connections from NS load balancers
allow_open_relay	bool	Allow connections to ports other than 80/443. DANGEROUS.
allow_rest_query	bool	Allows proxy queries. DANGEROUS.
always_do_dns	bool	Always do our own DNS lookup even if we could avoid it by trusting the client
append_query_params	string	Append (or update if already exist) specified query params to each request. Format: key1=val1&key2=val2

asymmetric_mode	bool	Enables diverting only symmetric traffic (requires divert_helper module installed)
blacklist_location	string	URL to download static filter URL list from
blacklist_refresh	int	List refresh period. Use 0 to disable auto refresh, list loaded on proxy start only
block_connection	bool	If set, block the connection
brightcloud_category_refresh	int	Period (in seconds) to wait before querying the update service since last update
brightcloud_deny_unknown	bool	URLs that are not yet categorised are classified Unkown
brightcloud_device	string	Brightcloud device
brightcloud_heartbeat	int	Heartbeat period (in seconds) with category update service
brightcloud_http_wrapper	bool	Whether requests to the update server should be formatted as an HTTP request
brightcloud_list	checklist	List of categories to be filtered separated by space or comma
brightcloud_max_validity	int	Period (in seconds) to wait before forcing a category refresh since last update
brightcloud_oem	string	Brightcloud OEM
brightcloud_port	int, r80-65535	The target port on the update server to connect to
brightcloud_queue_size	int	The outgoing requests queue size to the update service
brightcloud_recheck_timeout	int	Period (in seconds) to wait before querying the update service when information was not available
brightcloud_redirect_url	string	URL to redirect end users to if a URL matches a category in the dynamic filter list
brightcloud_retry_timeout	int	Period (in seconds) to wait before querying the update service for the same URL
brightcloud_server	string	Host name or IP address of the category update server
brightcloud_wait_remote	bool	If the URL is not found in the local category database we wait until is retrived remotely (timeout_filter controls for how long)
bypass_connection_count	int	New requests will have filtering and policies applied, but will not use the cache
bypass_cpu_level	int	New requests will have filtering and policies applied, but will not use the cache
cache_always	bool	If set, always cache the file even if headers don't allow
cache_async_fetch	bool	If set enables async fetch to cache. This means request with slight

		modifications is sent to the server asynchronously one more time to fetch the response and save it. Request modifications include stripping of some headers leading to not full response being served, e.g. Range. This can be useful if some client uses Range requests extensively
cache_async_refresh	bool	If set enables async refresh of expired cache items.This means proxy starts to serve cached response to client immediately while refreshing content in the background. This can lead to serving of stale content.
cache_clean_mode	string	Valid options: recursive/last_used/last_used_rmdir
cache_config_applies	bool	Enable / disable config apply caching optimisation for policy matching
cache_control	string	Override server's Cache-Control HTTP Header with this value
cache_db	string	Cache database pathname
cache_db_size	int	Size of the cache database in entries (each entry=8 bytes; 0 = autosize; -1 = disabled)
cache_delay	int	Sets how many times we need to have an object requested before we cache it
cache_disk_error_codes	string	The read error codes
cache_disk_error_rate	int	The rate of errors/sec when the disk is considered damaged (0 disables the checks)
cache_disks	string	List of cache disk mount points
cache_efficiency_threshold	int	Cache efficiency (byte hit rate) alert threshold (%)
cache_fast_disk_object_size	int	Maximum size of object to store in the fast disk cache
cache_ignore_cnc	bool	If set, ignore client no-cache headers (e.g. reload)
cache_index	string	Set the cache-index
cache_invalidate_on_get_with_body	bool	Invalidate cache item if a GET request for this item has body
cache_max_usage	int, r11-95	Max % of cache disk to use
cache_mem_model	string	The model used for RAM cache (index or mru)
cache_mem_object_size	int	Maximum size of object to store in the RAM cache
cache_mem_size	int	Maximum number of objects to store in the RAM cache
cache_never	bool	If set, never cache the file
cache_parser_delay	int	The delay in milliseconds between two disks operations performed by the

		background cache content parser.-1 will disable the background cache parsing.
cache_partial_checksum_start	int	Start offset for computing validation checksum for partial caching. Valid range: 0-32767
cache_partial_chunk_size	int	Chunk size for partial caching
cache_partial_download	bool	If set will allow caching partial requests and interrupted transfers
cache_post	bool	Force caching of HTTP POST responses
cache_ttl	int	Force a TTL (in seconds) for the object before revalidation
cache_x_headers	bool	Cache non-standard HTTP headers starting with "X-"
chassis_fan_speed_threshold	int	Chassis fan speed alert threshold (rpm)
cluster	iplist	List of IP addresses in the cluster
cluster_sync_fail_threshold	int	Number of failed cluster syncs to happen in a row before an alert is raised
complete_download	bool	Complete download to cache even if client aborted it
complete_download_min_count	int	Complete download if request is seen at least specified number of times (not applicable for sparse files and memory cache). Setting it to 0 will disable this trigger.
complete_download_threshold	int	Complete download if threshold is reached (% of full size)
compute_c32_sum	bool	Compute C32 checksum for access log field %C
connect_retry_attempts	int	A connect might fail to a valid endpoint due to fast port reuse. This setting allow a variable number of retries
content_hash	bool	Support content based hashing to index content
cpu_fan_speed_threshold	int	CPU fan speed alert threshold (rpm)
cpu_temp_threshold	int	CPU temperature alert threshold (degrees)
cpu_usage_threshold	int	CPU usage alert threshold (%)
debuglevel	int, r1-5	The debug level: 1=Errors, 2=Warnings, 3=Info, 4=Debug, 5=Full trace
default_deny	bool	If set all requests will be blocked except those enabled by the bypass list
default_ttl	int	Apply a TTL (in seconds) for responses without TTL properties.Special values: -1 - do not cache such objects, 0 - always revalidate
defer_accept	bool	Enable / disable tcp_defer_accept on the listening socket

delay_explicit_check	bool	Delay checking for explicit connection until client headers are read
dhcp	bool	Enable DHCP
disable_gzip_encoding	bool	If set will disable on the fly gzip content encoding for servers that send only identity encoding when the client supports gzip
disable_gzip_rebuild	bool	If set will disable gzip encoding for content already stored as identity when client supporting gzip request it
disable_swiftserve_auth	bool	Disable swiftserve auth
disks_usage_threshold	int	Disks usage alert threshold (%)
disks_util_threshold	int	Disks utilisation alert threshold (%)
dns_servers	iplist	List of DNS nameservers to use
dnscache_enabled	bool	Use a local DNS cache
dnscache_resolve_timeout	int	The number of seconds after which a resolve is considered to have failed
dnscache_size	int	The maximum number of entries we will keep in our own dns cache
dscp_client_hit	int	DSCP bits to be used for proxy->client [hit] traffic
dscp_client_miss	int	DSCP bits to be used for proxy->client [miss] traffic
dscp_server	int	DSCP bits to be used for proxy->server traffic
dump_headers	bool	Flag indicating should we dump headers to logs or not
effective_debuglevel	int, r1-5	Effective debug level. Should not be altered manually
effective_enable_logging	bool	Enable or disable logging completely. Should not be altered manually
enable_ipv6	bool	Enable IPv6 support on the interface
enable_logging	bool	Enable or disable logging completely. E.g. error, access logs
enable_ssl_proxy	bool	If set, enables SSL proxy support
enable_via	bool	If set proxy will add/update Via header in response
enable_x_cache	bool	If set, sends a x-cache header to the client
enable_x_cache_debug	bool	If set, sends a X-Cache-Debug header to the client. NOTE: This header contains internal information and may pose a security risk.

enable_x_forwarded	bool	If set, sends a x-forwarded-for header to the server
enabled	bool	Enable the interface
fast_disks	string	List of fast cache disk mount points for some content (leave this blank if all the cache disks have the same speed)
filter_path	string	Path where whitelist and blacklist filters stored
filter_rewrite_on_redirect	bool	If one of the filters applied by the policy triggers a redirect, rewrite the URL instead of redirecting. This means the cache will silently rewrite the URL instead of issuing a 302 redirect response. Use this if you don't want the user to see the URL change in their browser
filter_set	string	Names to identify the filter sets to be applied in the policy, if not present filtering will be disabled
force_relay	bool	If set, forces the connection into relay mode
gateway	ipv4	Default network gateway (ipv4)
gateway_mode	bool	The proxy will allow explicit requests from ipv4 clients to ipv6 server and vice-versa
global_blacklist_location	string	URL (starts with http://) or local file path (starts with /) to download list of global block/redirect URLs from
global_blacklist_refresh	int	List refresh period. Use 0 to disable auto refresh, list loaded on proxy start only
global_redirect_url	string	URL (starts with http://) or local file path (starts with /) to redirect on global block list hits, if empty the request will be blocked
global_whitelist_location	string	URL (starts with http://) or local file path (starts with /) to download list of global bypass URLs from
global_whitelist_refresh	int	List refresh period. Use 0 to disable auto refresh, list loaded on proxy start only
group	string	Group that proxy runs as
hostname	string	System hostname
icap_respmod	bool	Process server responses using RESPMOD ICAP request
icap_server	string	ICAP server URL (icap://host:port/uri)
icc_enabled	bool	Enable inter-cache communication (ICC)
icc_min_size	int	Minumum object size for inter-cache communication
icc_override_enabled	bool	Enable ICC in policy

icc_relocation_enabled	bool	Enable relocation of objects after ICC reconfigurantion
ignore_range	bool	Strip Range header from request when sending to server. Responses from cache will still obey Range header
io_queue_size	int	Maximum async I/O queue size for each disk. 0 - unlimited.
io_threads_per_disk	int	Threads per disk for async disk I/O
ip	ipv4	IPv4 address of the network interface
ip_spoofing	bool	Enable fully transparent deployment (client IP address spoofing)
ipv6_addr	ipv6	IPv6 address of the network interface
ipv6_gateway	ipv6	Default network gateway (ipv6)
ipv6_netmask	int, r1-128	Netmask (ipv6) for the interface
log_debug_modules	int	Debug/trace log modules. Bitwise OR of module IDs. Module IDs: MODULE_MASK_AUTH 0x00000001 MODULE_MASK_CACHE 0x00000002 MODULE_MASK_CACHECOM 0x00000004 MODULE_MASK_CONFIG 0x00000008 MODULE_MASK_DNS 0x00000010 MODULE_MASK_FILTER 0x00000020 MODULE_MASK_HTTP 0x00000040 MODULE_MASK_LICENSE 0x00000080 MODULE_MASK_LOGGING 0x00000100 MODULE_MASK_STREAMING 0x00000200 MODULE_MASK_NET 0x00000400 MODULE_MASK_PROXY 0x00000800 MODULE_MASK_SPM 0x00001000 MODULE_MASK_UTIL 0x00002000 MODULE_MASK_PROXY_MODES 0x00004000 MODULE_MASK_FILE_FORMAT 0x00008000 MODULE_MASK_SSL 0x00010000 MODULE_MASK_ALL 0xFFFFFFFF
log_location	string	Path to write logs to
log_location_size	int	Size of log partition in Mbytes. Negative means auto-detect
log_retention_period	int, ~+	Maximum age of retained log files (days). Zero means never delete
log_retention_threshold	int, r0-100	Percentage of log partition size to retain log files
log_rotation_period	float, ~+	Maximum age of an individual log file after which it should be rotated (hours)
log_rotation_schedule	log_rotation, ~:/(,)*	Log Rotation Schedule. Format: "%H:%M/%d,%d...", where %H and %M are hour and minute on day respectively to rotate logs on and %d is day number to rotate on (1 is Monday). So, to rotate at 11:45 PM each day except Wednesday this field should have value "23:45/1,2,4,5,6,7"
log_rotation_size	int, ~+	Maximum size of an individual log file after which it should be rotated (MB)
log_rotation_type	combo	Type of log rotation schedule: "p" for periodic, "s" for size, "d" for schedule

		(days)
log_upload_enable	bool	Flag to specify whether logs should be uploaded
log_upload_files	string	Comma-separated list of file names to upload. May use shell patterns. Allowed patterns are *,?,[seq],[!seq]. E.g. access*.gz Empty value means upload everything. Log files has following format <logname>-<hostname>-<starttime>-<endtime>.log.gz
log_upload_password	password	Password to authenticate against remote server
log_upload_path	string	Path on remote server to upload log files to
log_upload_protocol	combo	Protocol which should be used to upload compressed log files (FTP, SFTP or FTPs)
log_upload_server	string	Fully qualified server name to upload log files to
log_upload_user	, ~[.][\w\.\-]+	Username to authenticate against log server
log_usage_limit_threshold	int, r0-100	Max usage threshold (%) of log partition after which logging is disabled
log_usage_warning_clean_threshold	int, r0-100	Percentage of log partition size to delete old log files to if warning threshold is reached. Ignored if other option than "Delete old logs" specified
log_usage_warning_threshold	int, r0-100	Warning threshold (%) of log partition usage
log_usage_warning_type	combo	Flag indicating what to do if log usage is more than log usage warning threshold. Valid values are: 'i' - ignore, 'd' - delete old logs, 'r' - reduce log level
master	combo	Master interface. Use to make current interface a member of a bridge or interface bonding. If this is set to something other than None then some settings like ip address/net mask are ignored. This setting is ignored for non-ethernet interfaces.
max_cache_object_size	int	Max response size to cache. Larger responses will not be cached and will use the CLARGE flag in log entries
max_ttl	int	Force revalidation after max_ttl seconds after the last validation.Special values: 0 - disable Max TTL
maxfd	int	Max number of file descriptors
mb_temp_threshold	int	Motherboard temperature alert threshold (degrees)
mem_buffers_threshold	int	Memory buffers alert threshold (%)
memory_usage_threshold	int	Memory usage alert threshold (%)

mgmt_module_logging	string	JSON-encoded dictionary of logging levels for mgmt app. Do not edit unless you know what you're doing
monitor_password	password	Password for the monitor user
netmask	ipv4	Netmask (ipv4) for the interface
network_buffer_size	int, r1024-131072	The network buffer size
ntp_servers	string	Comma-separated list of NTP servers to use
origin	string	Request host address will be substituted with this (reverse proxy mode)
origin_bad_ssl_behavior	string	Defines the behavior when the origin endpoint doesn't provide a valid SSL certificate. Accepted values: gateway_error, drop_connection, redirect, ignore
origin_host	string	Host header for reverse proxy requests
packet_drop_received_threshold	float, r0-100	Received packets drop threshold (%)
packet_drop_sent_threshold	float, r0-100	Sent packets drop threshold (%)
password_bypass_key	string	A 16 chars [128 bits] secret key that will be used for cookie encryption. If empty, encryption will be disabled
password_bypass_url	string	The url that will set the password bypass cookie, if empty the bypass will be disabled
port	int, r80-65535	Port that proxy should bind to
proxy_auth_host	string	The host where the auth daemon is running.
proxy_auth_message	string	The message sent to the client when auth is needed
proxy_auth_port	int	The port where the auth dameon is running.
proxy_auth_required	bool	Explicit request are allowed if and only if the client sends valid auth info.
proxy_throughput_threshold	int	Cache throughput alert threshold (%), 0 means turn off this alert
purge_older_than	string	Purge cache objects older than specified date. Date format is RFC-822/850 or YYYY-MM-DD HH:MI:SS (GMT)
pversion	string	Policy version, should not be altered
ratelimit	int	Enables client connection throttling. Default: 0 - no throttling
ratelimit_burst	int	Apply client connection throttling only after specified size of data has been

		sent.
ratelimit_type	string	Specifies client connection throttling type: ip, connection. Default: ip - per-IP throttling
redirect	string	Redirect connection to this URL (via 302 redirect)
relay_connection_count	int	The proxy will act a traffic bridge when in intercept/transparent mode, no filtering, server/url policies will be applied.
relay_cpu_level	int	The proxy will act a traffic bridge when in intercept/transparent mode, no filtering, server/url policies will be applied.
relay_malformed_requests	bool	Start relay mode if failed to parse client request
return_to_sender	bool	Enable / disable return to sender mode
routing	multiline	Static routing rules for the interface. Use with caution. Format is "X.X.X.X/X via X.X.X.X dev IFACE" per line
rtmp_cache_db	string	RTMP cache database pathname
rtmp_dump_data	bool	Dump RTMP data
rtmp_port	int, r80-65535	RTMP Proxy Port
rtmp_stats_port	int, r80-65535	RTMP Stats Port
rtmp_tproxy_mode	bool	Transparent Proxy Mode
rtmp_whitelist	string	Filename with list of wildcard URLs allowed to be cached by rtmp proxy
section_disabled	bool	Disable this policy
seek_offset	string	Key that contains video seek offset
seek_range	string	Some sites (typically using flash) send URL arguments that are to be treated as HTTP range arguments. This option allows to set custom Range value to address such cases. The value can contain capture variables and should be evaluated to byte offsets delimited by a dash. For example: "*range$_{start}$* – range_end" where range_start and range_end are names of policy matches capture variables
seek_range_param	string	Query string param holding seek range
seek_strategy	combo	Video seek subtype for FLV video
seek_time_factor	float	Multiplier for seek offset to convert it to seconds

seek_type	combo	Video container type: FLV, MPEG4 and Range are currently supported. For Range type, use Seek Range field to specify the range
service_quality_alert_enable	bool	Enable service quality monitoring by weighted average TTFB across top 100 sites
service_quality_alert_serious_threshold	int	Service quality serious alert threshold
service_quality_alert_warning_threshold	int	Service quality warning alert threshold
service_time_threshold	int	Service time alert threshold (%), 0 means turn off this alert
share_server_connection	bool	Use single server connection for the same urls
smtp_from	string	SMTP From header
smtp_host	string	SMTP server host to use to send emails
smtp_password	string	Password to authenticate on SMTP server against
smtp_port	int, r1-65535	Port to use to connect to SMTP server
smtp_return_path	string	SMTP Return-Path header
smtp_ssl	bool	Use SSL to connect to SMTP server
smtp_username	string	User name to authenticate on SMTP server against. Empty means no authentication required
snmp_community	string	SNMP community name. Leave blank to set it to default 'public'.
spm_agents_url	string	The url (or file path) where the user-agent definitionlist is located
spm_enabled	bool	If set, enabled spm support
spm_encryption_key	string	A 16 chars [128 bits] secret key that will be used for encryptionwhen sending info to the redirect url
spm_heartbeat_timeout	int	The maximum duration to receive a heartbeat message
spm_policy_url	string	The url to retrive policy definsitions frommust start with http://, ID will be replaced with the actual policy id [ie: http://spm.server.com/policy/code1/ID/code2/ID]
spm_sessions_url	string	The url to retrieve the list of active sessions
spm_stomp_pass	string	The password used to connect to the apachemq
spm_stomp_url	string	The apachemq stomp address where the updates will be pushedin

		stomp://server.address.com/queue_name
spm_stomp_user	string	The username used to connect to the apachemq
ssl_ca_bundle_path	int	The CA bundle to be used for server certs verification. If empty the openssl default one will be used
ssl_cache_cert_path	string	If set then on-disk storage of generated SSL certificates will be enabled, this setting defines the cert storage dir. If not set then certificates will be cached in memory only.
ssl_cache_cert_size	int	The maximum number of SSL certificates we cache in memory
ssl_use_existing_cert	string	If not empty should contain the path to an existing certificate/key file to be used instead of the generated ones
static_redirect_url	string	URL to redirect end users to if a URL in the static filter list is matched
super_cluster	iplist	List of IP addresses in super cluster
super_cluster_secret	password	Secret used for internal communication in the super-cluster. Must be the same on all machines in the super-cluster
swap_usage_threshold	int	Swap usage alert threshold (%)
swiftsense_alertsreport_enable	bool	Enable SwiftSense alerts report feature. This will send all new alerts to SwiftSense and you'll be able to have better knowledge of what's wrong with you caches
swiftsense_auth_cookie	string	
swiftsense_configreporter_enable	bool	Enable sending current config to SwiftSense. This allows to view config values / policies for the cache in SwiftSense.
swiftsense_enable	bool	Enable Policy Center feature
swiftsense_licenseupdate_enable	bool	Enable license keys update
swiftsense_policies_set	string	Swiftsense policies set
swiftsense_policyupdate_enable	bool	Enable Policy Center policy update feature. This will ensure that you have latest versions of site-specific policies to improve cache hit rates
swiftsense_policyupdate_upgrade_enable	bool	Enable Policy Center policy upgrade feature. This will upgrade all not modified policies after update them from Policy Center
swiftsense_rpmupdate_enable	bool	Enable automatic fetching of available updates list. This allows you to easily upgrade your SC installation/cluster
swiftsense_server	string	Policy Center server address

swiftsense_top100_enable	bool	Enable Policy Center top 100 reporting. This will send top 100 report to the Policy Controller periodically so we can enhance policies basing on this data
swiftserve_enable	bool	Enable the delivery of traffic on behalf of the SwiftServe Content Delivery Network
swiftserve_enable_status	bool	Enable Swiftserve status page
swiftserve_log_location	string	Directory for swiftserve logs
swiftserve_mode	bool	Apply swiftserve policies and send logs to swiftserve
swiftserve_node_reporting_id	string	Reporting id for swiftserve
taskmanager_control_sock_path	string	Control socket for taskmanager. If empty then task manager control will be disabled
tcp_orphans_threshold	int	TCP orphans alert threshold (%)
tcp_tw_buckets_threshold	int	TCP time-wait sockets alert threshold (%)
threads	int, r0-128	Number of worker threads; 0 means auto-detect
timeout_auth_daemon	int	Timeout (in seconds) to wait for a response from the auth daemon
timeout_client_read	int	Timeout (in seconds) when reading from a client
timeout_client_start	int	Timeout (in seconds) when reading from a client before forcing relay mode
timeout_client_write	int	Timeout (in seconds) when writing to a client
timeout_connect	int	Timeout (in seconds) before we assume that a connect has failed
timeout_filter	int	Timeout (in seconds) to wait for a filtering decision
timeout_keep_alive	int	Timeout (in seconds) to wait for a new request over KA connections
timeout_relay	int	Timeout (in seconds) before we assume that a relay connection is dead
timeout_server_read	int	Timeout (in seconds) when reading from a server
timeout_server_write	int	Timeout (in seconds) when writing to a server
timeout_ssl_handshake	int	Timeout (in seconds) to wait for a SSL handshake to complete
timezone	combo	System timezone
top100_check_interval	int	Interval in seconds between top100 data parsing attempts. Needs task manager restart on change.

top100_delay	int	Delay in seconds between two parsing rounds
top100_enabled	bool	Enable access.log parsing to generate top100 reports. Can be cpu-consuming. NOTE: you need to restart task manager after changing this setting
top100_maxtime	int	Max time in seconds to do one parsing round for
top100_maxtries	int	Max count of parsing rounds to do for one parsing attempt
top100_min_entries	int	Min number of records to keep in stats frame during stats trimming
top100_min_percentage	int	Min percentage of traffic to trim stats record to
top100_parse_failure_alert_threshold	int, r0-100	
top100_video_regexes	string	Comma-separated list of video url patterns in format domain/regex; regex should contain a capture named id matching video ID. Example: youtube.com/watch?(|.*&)v=(?P<id>[^&]+).
tproxy_mode	bool	Enable / disable tproxy compatible mode
trust_x_forwarded	bool	Trust x-forwarded-for headers in request
url	string	Rewrite request URL
url_prefix_optimisation	bool	Enable / disable URL prefix optimisation for policy matching
user	string	User that proxy runs as
whitelist_location	string	URL to download bypass URL list from
whitelist_refresh	int	List refresh period. Use 0 to disable auto refresh, list loaded on proxy start only